

Guida per gli sviluppatori

AWS AppSync GraphQL



AWS AppSync GraphQL: Guida per gli sviluppatori

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

I marchi e il trade dress di Amazon non possono essere utilizzati in relazione ad alcun prodotto o servizio che non sia di Amazon, in alcun modo che possa causare confusione tra i clienti, né in alcun modo che possa denigrare o screditare Amazon. Tutti gli altri marchi non di proprietà di Amazon sono di proprietà dei rispettivi proprietari, che possono o meno essere affiliati, collegati o sponsorizzati da Amazon.

Table of Contents

Che cos'è AWS AppSync	1
AWS AppSync features	1
Sei un AWS AppSync utente alle prime armi?	2
Servizi correlati	2
Prezzi per AWS AppSync	3
GraphQL e architettura AWS AppSync	4
Che cos'è un'API	5
Client	5
Resources	5
Che cos'è REST	6
Interfaccia uniforme	6
Apolidia	7
Sistema a strati	7
Cacheabilità	7
Che cos'è un'API? RESTful	7
Come RESTful APIs funziona?	8
Che cos'è GraphQL	8
Confronto tra REST e GraphQL	8
Perché usare GraphQL rispetto a REST	11
Componenti di un'API GraphQL	12
Schemi GraphQL	13
Fonti di dati	32
Risolutori	47
Proprietà aggiuntive di GraphQL	57
Dichiarativo	57
Gerarchico	57
Introspettiva	59
Digitazione forte	60
Guida introduttiva: creazione della prima API GraphQL	61
Avvio di uno schema	62
Facciamo un giro della console AWS AppSync	66
Progettista dello schema	66
Origini dati	67
Query	68

Settings	68
Utilizzo delle mutazioni GraphQL per aggiungere dati a una tabella DynamoDB	69
Utilizzo delle query GraphQL per recuperare dati da una tabella DynamoDB	74
Sezioni supplementari	77
Integrazione	77
Lettura supplementare	78
Progettazione GraphQL APIs	79
Strutturazione di un'API GraphQL (vuota o importata) APIs	79
Progettazione dello schema GraphQL	80
Allegare una fonte di dati	108
Configurazione AWS AppSync dei resolver	120
Utilizzo APIs con il CDK	176
Utilizzo di abbonamenti per applicazioni di dati in tempo reale	194
Direttive di sottoscrizione allo schema GraphQL	194
Utilizzo degli argomenti di sottoscrizione	197
Creazione di prodotti generici pub/sub APIs con tecnologia serverless WebSockets	201
Definizione di filtri avanzati per gli abbonamenti	205
Annullamento della sottoscrizione delle connessioni utilizzando WebSocket i filtri	216
Creazione di un WebSocket client in tempo reale	220
Fusione APIs	242
Unione e federazione APIs	244
Risoluzione unificata dei conflitti tramite API	246
Configurazione degli schemi	254
Configurazione delle modalità di autorizzazione	254
Configurazione dei ruoli di esecuzione	255
Configurazione della fusione tra account tramite APIs AWS RAM	257
Unire	258
Supporto aggiuntivo per Merged APIs	260
Limitazioni delle API unite	261
Considerazioni sull'API unita	261
Creazione di un sistema unito APIs	262
Creazione di GraphQL APIs con l'introspezione RDS	263
Utilizzo della funzione di introspezione (console)	264
Utilizzo della funzione di introspezione (API)	268
Creazione di un'applicazione client	271
JavaScript tutorial sui resolver	274

Creazione di una semplice applicazione post utilizzando i resolver DynamoDB JavaScript	274
Creazione dell'API GraphQL	275
Definizione di un'API post di base	275
Configurazione della tabella Amazon DynamoDB	276
Configurazione di un resolver AddPost (Amazon DynamoDB) PutItem	277
Configurazione del resolver GetPost (Amazon DynamoDB) GetItem	280
Creare una mutazione UpdatePost (Amazon DynamoDB) UpdateItem	283
Crea mutazioni di voto (Amazon DynamoDB UpdateItem)	288
Configurazione di un resolver DeletePost (Amazon DynamoDB) DeleteItem	291
Configurazione di un resolver AllPost (Amazon DynamoDB Scan)	297
Configurazione di un allPostsBy Author resolver (Amazon DynamoDB Query)	302
Utilizzo dei set	307
Conclusioni	314
AWS Lambda Usare i resolver	314
Powertools per AWS Lambda	314
Creazione di una funzione Lambda	315
Configurare un'origine dati per Lambda	316
Creare uno schema GraphQL	317
Configura i resolver	121
Testa la tua API GraphQL	319
Errori di restituzione	321
Caso d'uso avanzato: Batching	324
Utilizzo di resolver locali	333
Creazione dell'app pub/sub	333
Invia e sottoscrivi messaggi	334
Combinazione di resolver GraphQL	335
Schema di esempio	336
Alterazione dei dati tramite resolver	337
DynamoDB e Service OpenSearch	337
Utilizzo dei resolver di servizio OpenSearch	340
Crea un nuovo dominio di servizio OpenSearch	340
Configura una fonte di dati per Service OpenSearch	341
Collegamento di un resolver	342
Modificare le ricerche	343
Aggiungere dati al servizio OpenSearch	345
Recupero di un singolo documento	346

Esegui interrogazioni e mutazioni	347
Best practice	347
Esecuzione di transazioni DynamoDB	348
Permissions	348
Origine dati	349
Transazioni	351
Utilizzo delle operazioni batch di DynamoDB	358
Batch a tabella singola	358
Batch multitavola	363
Gestione degli errori	371
Utilizzo di resolver HTTP	377
Creazione di un'API REST	377
Creazione dell'API GraphQL	378
Creazione di uno schema GraphQL	378
Configura la tua fonte di dati HTTP	379
Configurazione dei resolver	153
Richiamo dei servizi AWS	383
Usare Aurora PostgreSQL con Data API	384
Configura il tuo database Aurora PostgreSQL	385
Creazione del database e della tabella	386
Creazione di uno schema GraphQL	387
Resolver per RDS	389
Eliminazione del cluster	397
Tutorial sui resolver VTL	398
Creazione di una semplice applicazione post utilizzando i resolver DynamoDB	399
Configurazione delle tabelle DynamoDB	399
Creazione dell'API GraphQL	378
Definizione di una post API di base	401
Configurazione dell'origine dati per le tabelle DynamoDB	402
Configurazione del resolver AddPost (DynamoDB) PutItem	403
Configurazione del GetPost Resolver (DynamoDB) GetItem	408
Creare una mutazione UpdatePost (DynamoDB) UpdateItem	411
Modifica del Resolver UpdatePost (DynamoDB) UpdateItem	415
Crea mutazioni UpvotePost e DownvotePost (DynamoDB) UpdateItem	421
Configurazione del DeletePost Resolver (DynamoDB) DeleteItem	425
Configurazione del resolver Up the allPost (scansione di DynamoDB)	432

Configurazione dell' allPostsByAuthor Resolver (DynamoDB Query)	437
Uso di set	307
Uso di elenchi e mappe	450
Conclusioni	454
AWS Lambda Usare i resolver	454
Creazione di una funzione Lambda	454
Configurare un'origine dati per Lambda	456
Creare uno schema GraphQL	378
Configura i resolver	153
Testa la tua API GraphQL	461
Errori di restituzione	462
Caso d'uso avanzato: Batching	465
Utilizzo dei resolver di servizio OpenSearch	475
Impostazione One-Click	476
Crea un nuovo dominio di servizio OpenSearch	476
Configura l'origine dei dati per il servizio OpenSearch	476
Collegamento di un Resolver	478
Modificare le ricerche	480
Aggiungere dati al servizio OpenSearch	482
Recupero di un documento individuale	483
Eseguire query e mutazioni	483
Best practice	484
Utilizzo di resolver locali	484
Creare l'applicazione di paging	485
Inviare e iscriversi alle pagine	486
Combinazione di resolver GraphQL	487
Schema di esempio	487
Modificare i dati tramite resolver	488
DynamoDB e Service OpenSearch	489
Utilizzo delle operazioni batch di DynamoDB	493
Permissions	494
Origine dati	495
Batch a tabella singola	495
Batch a tabella multipla	499
Gestione errori	507
Esecuzione di transazioni DynamoDB	513

Permissions	494
Origine dati	495
Transazioni	515
Utilizzo di resolver HTTP	525
Impostazione One-Click	476
Creazione di un'API REST	377
Creazione dell'API GraphQL	378
Creazione di uno schema GraphQL	378
Configurazione dell'origine dati HTTP	379
Configurazione dei resolver	153
Richiamo dei servizi AWS	531
Utilizzo dei resolver Aurora Serverless v2	533
Configurazione del cluster di database	534
Abilitazione dell'API di dati	535
Creazione di un database e di una tabella	535
Schema GraphQL	536
Connect la tua API alle operazioni del database	153
Modifica i tuoi dati tramite l'API	545
Recupera i tuoi dati	547
Proteggi l'accesso ai dati	548
Utilizzo dei resolver per pipeline	550
Impostazione One-Click	476
Configurazione manuale	551
Test dell'API GraphQL	461
Utilizzo delle operazioni Delta Sync su fonti di dati con versioni	565
Impostazione One-Click	476
Schema	566
Mutazioni	569
Query di sincronizzazione	569
Esempio	569
Configurazione e impostazioni	576
Configurazione della memorizzazione nella cache lato server e della compressione del payload delle API	577
Tipi di istanza	577
Comportamento della cache	578
Crittografia cache	581

Eliminazione della cache	581
Eliminare una voce dalla cache	581
Eliminare una voce della cache in base all'identità	583
Compressione delle risposte API	585
Configurazione di nomi di dominio personalizzati per GraphQL e in tempo reale APIs	585
Registrazione e configurazione di un nome di dominio	586
Creazione di un nome di dominio personalizzato in AWS AppSync	586
Nomi di dominio personalizzati Wildcard in AWS AppSync	588
Operazioni di controllo delle versioni, rilevamento dei conflitti e sincronizzazione per DynamoDB	588
Controllo delle versioni delle sorgenti dati DynamoDB	589
Rilevamento e risoluzione dei conflitti	593
Utilizzo delle operazioni di sincronizzazione di DynamoDB su sorgenti dati con versioni	604
Utilizzo CloudWatch per monitorare e registrare i dati dell'API GraphQL	604
Installazione e configurazione	605
CloudWatch metriche	607
CloudWatch registri	618
Riferimento al tipo di registro	623
Analisi CloudWatch dei log con Logs Insights	625
Analizza i tuoi log con Service OpenSearch	627
Migrazione del formato di registro	627
Tracciamento delle richieste in AWS X-Ray	628
Installazione e configurazione	605
Tracciamento delle API con X-Ray	629
Registrazione delle chiamate API di AWS CloudTrail con	631
AWS AppSync informazioni in CloudTrail	631
AWS AppSync eventi di dati in CloudTrail	632
Comprendere le AWS AppSync voci dei file di registro	634
Utilizzo di AWS AppSync Private APIs	641
Creazione di contenuti privati AWS AppSync APIs	643
Creazione di endpoint di interfaccia per AWS AppSync	644
Esempi avanzati di	646
Esempi di piani di controllo	650
Utilizzo delle policy IAM per limitare la creazione di API pubbliche	651
Supporto VPC PrivateLink	652
Condivisione di GraphQL APIs	652

Prerequisiti per la condivisione AWS AppSync di GraphQL APIs	653
Condividi AWS AppSync GraphQL APIs	653
Smetti di condividere AWS AppSync GraphQL APIs	655
Eventi multi-account	656
Configurazione della complessità dell'esecuzione, della profondità delle query e dell'introspezione di GraphQL	656
Utilizzo della funzione di introspezione	656
Configurazione dei limiti di profondità delle interrogazioni	658
Configurazione dei limiti di conteggio dei resolver	659
Utilizzo delle variabili di ambiente	661
Configurazione delle variabili di ambiente (console)	662
Configurazione delle variabili di ambiente (API)	663
Configurazione delle variabili di ambiente (CFN)	664
variabili di ambiente e unite APIs	664
Recupero di variabili di ambiente	664
Autorizzazione e autenticazione GraphQL APIs	667
Tipi di autorizzazione	667
API_KEY autorizzazione	668
AWS_LAMBDA autorizzazione	670
Eludere le limitazioni di autorizzazione dei token SigV4 e OIDC	676
AWS_IAM autorizzazione	677
OPENID_CONNECT autorizzazione	679
AMAZON_COGNITO_USER_POOLS autorizzazione	681
Utilizzo di modalità di autorizzazione aggiuntive	682
Controllo granulare degli accessi	685
Filtraggio delle informazioni	688
Accesso origine dati	689
Casi d'uso del controllo degli accessi per proteggere richieste e risposte	689
Panoramica di	690
Lettura dei dati	690
Scrittura di dati	694
Record pubblici e privati	697
Dati in tempo reale	698
AWS WAF Usare per proteggere APIs	701
Integra un'API con AppSync AWS WAF	702
Creazione di regole per un ACL web	704

Sicurezza	708
Protezione dei dati	709
Crittografia in movimento	710
Convalida della conformità	710
Sicurezza dell'infrastruttura	710
Resilienza	711
Gestione dell'identità e degli accessi	711
Destinatari	712
Autenticazione con identità	712
Gestione dell'accesso tramite policy	713
Come AWS AppSync funziona con IAM	715
Policy basate sull'identità	721
Risoluzione dei problemi	733
Registrazione delle chiamate AWS AppSync API con AWS CloudTrail	735
AWS AppSync informazioni in CloudTrail	736
Comprensione delle AWS AppSync voci dei file di registro	737
Best practice	484
Comprendi i metodi di autenticazione	739
Scopri come si propagano le modifiche alla configurazione delle API	740
Usa TLS per i resolver HTTP	740
Usa ruoli con il minor numero di autorizzazioni possibile	740
Le migliori pratiche in materia di policy IAM	740
Riferimento al resolver () JavaScript	742
JavaScript panoramica dei resolver	742
Funzionalità di runtime supportate	743
Risolutori di unità	743
Anatomia di un resolver di pipeline JavaScript	743
Argomenti supplementari	748
Esempio di risolutore di pipeline con Amazon DynamoDB	748
Configurazione delle utilità per il runtime APPSYNC_JS	751
Raggruppamento TypeScript e mappe di origine per il runtime APPSYNC_JS	754
Test del resolver e dei gestori di funzioni	760
Migrazione da VTL a JavaScript	763
Scelta tra accesso diretto alla fonte di dati e invio di proxy tramite un'origine dati Lambda ...	766
JavaScript riferimento all'oggetto contestuale del resolver	768
Accesso a context	768

JavaScript caratteristiche di runtime per resolver e funzioni	779
Funzionalità di runtime supportate	780
Utilità integrate	787
Moduli incorporati	790
- Utilità di runtime	814
Aiutanti temporali in util.time	815
Aiutanti DynamoDB in util.dynamodb	817
Helper HTTP in util.http	823
Aiutanti di trasformazione in util.transform	825
String helper in util.str	839
Estensioni	839
Helper XML in util.xml	843
JavaScript riferimento alla funzione resolver per DynamoDB	845
GetItem	847
PutItem	848
UpdateItem	851
DeleteItem	856
Query	859
Scan	864
Sync	868
BatchGetItem	871
BatchDeleteItem	874
BatchPutItem	877
TransactGetItems	879
TransactWriteItems	882
Sistema di tipi (mappatura delle richieste)	889
Sistema di tipi (mappatura delle risposte)	894
Filtri	898
Espressioni di condizione	900
Espressioni relative alle condizioni delle transazioni	913
Proiezioni	916
JavaScript riferimento alla funzione resolver per OpenSearch	918
Richiesta	918
Risposta	919
Campo operation	919
Campo path	920

Campo params	920
Passare variabili	922
JavaScript riferimento alla funzione resolver per Lambda	923
Oggetto Request	923
Oggetto Response	927
Risposta in batch della funzione Lambda	927
JavaScript riferimento alla funzione resolver per la fonte di dati EventBridge	928
Richiesta	918
Risposta	928
PutEvents campi	930
JavaScript riferimento alla funzione resolver per la fonte di dati None	932
Richiesta	918
Carico utile	926
Risposta	928
JavaScript riferimento alla funzione resolver per HTTP	933
Richiesta	918
Metodo	934
ResourcePath	934
Campi Params	935
Risposta	928
JavaScript riferimento alla funzione resolver per Amazon RDS	936
Modello con tag SQL	937
Creazione di dichiarazioni	938
Recupero dei dati	938
Funzioni di utilità	939
Casting	947
JavaScript riferimento alla funzione resolver per Amazon Bedrock	949
Oggetto Request	949
Oggetto Response	956
Invocazioni di lunga durata	956
Riferimento del tipo	957
Riferimento al modello di mappatura Resolver (VTL)	964
Panoramica dei modelli di mappatura Resolver	965
Resolver di unità	965
Resolver per pipeline	171
Modello di esempio	970

Regole di deserializzazione dei modelli di mappatura valutate	972
Guida alla programmazione dei modelli di mappatura Resolver	974
Configurazione	975
Variabili	976
Metodi di chiamata	978
Stringhe	979
Loop	980
Matrici	981
Controlli condizionali	982
Operatori	983
Contesto	985
Filtraggio	985
riferimento al contesto del modello di mappatura Resolver	990
Accesso a \$context	990
Neutralizzazione degli input	1000
riferimento all'utilità del modello di mappatura Resolver	1002
Utility helper in \$util	1003
AWS AppSync direttive	1016
Aiutanti temporali in \$util.time	1017
Elenca gli aiutanti in \$util.list	1019
Aiutanti di mappe in \$util.map	1020
Helper DynamoDB in \$util.dynamodb	1021
Aiutanti Amazon RDS in \$util.rds	1031
Aiutanti HTTP in \$util.http	1034
Helper XML in \$util.xml	1036
Aiutanti di trasformazione in \$util.transform	1038
Aiutanti matematici in \$util.math	1052
Aiutanti per le stringhe in \$util.str	1053
Estensioni	1054
Riferimento al modello di mappatura dei resolver per DynamoDB	1068
GetItem	1070
PutItem	1072
UpdateItem	1075
DeleteItem	1081
Query	1084
Scan	1089

Sync	1094
BatchGetItem	1097
BatchDeleteItem	1101
BatchPutItem	1105
TransactGetItems	1108
TransactWriteItems	1112
Sistema di tipi (mappatura delle richieste)	1121
Sistema di tipi (mappatura delle risposte)	1126
Filtri	1130
Espressioni di condizione	1131
Espressioni relative alle condizioni delle transazioni	1144
Proiezioni	1147
Riferimento al modello di mappatura Resolver per RDS	1149
Richiedi un modello di mappatura	1149
Versione	1151
Dichiarazioni e VariableMap	1151
VariableTypeHintMap	1151
Riferimento al modello di mappatura Resolver per OpenSearch	1152
Richiedi un modello di mappatura	1149
Modello di mappatura delle risposte	919
Campo operation	919
Campo path	920
Campo params	920
Passare variabili	922
Riferimento al modello di mappatura Resolver per Lambda	1157
Richiedi un modello di mappatura	1149
Modello di mappatura delle risposte	919
Risposta in batch della funzione Lambda	1163
Resolver Lambda diretti	1163
Riferimento al modello di mappatura Resolver per EventBridge	1169
Richiedi un modello di mappatura	1149
Modello di mappatura delle risposte	919
PutEvents campi	930
Riferimento al modello di mappatura Resolver per l'origine dati None	1174
Richiedi un modello di mappatura	1149
Versione	1151

Carico utile	1161
Modello di mappatura delle risposte	919
Riferimento al modello di mappatura Resolver per HTTP	1177
Richiedi un modello di mappatura	1149
Versione	1151
Metodo	1180
ResourcePath	1180
Campi dei parametri	920
Risposta	1182
Autorità di certificazione (CA) riconosciute dagli AWS AppSync endpoint HTTPS	1182
log delle modifiche del modello di mappatura Resolver	1245
Disponibilità delle operazioni dell'origine dati per matrice di versione	1246
Modifica della versione su un modello di mappatura del resolver di unità.	1247
Modifica della versione su una funzione	1248
2018-05-29	1249
2017-02-28	1256
Riferimento al tipo GraphQL	1257
Tipi scalari in GraphQL	1258
Scalari predefiniti	1258
AWS AppSync scalari	1258
Esempio di utilizzo dello schema	1260
Interfacce e unioni in GraphQL	1263
Esempi di interfacce	1264
Esempi di unione	1268
Digita la risoluzione in AWS AppSync	1269
Esempio di risoluzione dei tipi	1269
Risoluzione dei problemi ed errori comuni	1275
Mappatura della chiave di DynamoDB errata	1275
Resolver mancante	1275
Errori del modello di mappatura	1276
Tipi restituiti non corretti	1276
Elaborazione di richieste non valide	1277
.....	mcclxxviii

Che cos'è AWS AppSync?

AWS AppSync consente agli sviluppatori di connettere le proprie applicazioni e servizi a dati ed eventi con GraphQL e Pub/Sub sicuri, serverless e ad alte prestazioni. APIs È possibile effettuare le seguenti operazioni con: AWS AppSync

- Accedi ai dati da una o più fonti di dati da un singolo endpoint dell'API GraphQL.
- Combina più sorgenti GraphQL APIs in un'unica API GraphQL unita.
- Pubblica aggiornamenti dei dati in tempo reale sulle tue applicazioni.
- Sfrutta la sicurezza, il monitoraggio, la registrazione e la tracciabilità integrati, con caching opzionale per una bassa latenza.
- Paghi solo per le richieste API e i messaggi in tempo reale che vengono recapitati.

Important

A partire dal 13 marzo 2025, puoi creare un' PubSub API in tempo reale basata su WebSockets AWS AppSync Events. Per ulteriori informazioni, consulta [Pubblicare eventi tramite WebSocket](#) nella AWS AppSync Events Developer Guide.

Argomenti

- [AWS AppSync features](#)
- [Sei un AWS AppSync utente alle prime armi?](#)
- [Servizi correlati](#)
- [Prezzi per AWS AppSync](#)

AWS AppSync features

- Accesso ai dati e interrogazione semplificati, con tecnologia GraphQL
- Serverless WebSockets per abbonamenti GraphQL e canali pub/sub
- Caching lato server per rendere disponibili i dati in cache in memoria ad alta velocità per una bassa latenza
- JavaScript e supporto per la scrittura di logiche aziendali TypeScript

- Sicurezza aziendale con Private APIs per limitare l'accesso alle API e l'integrazione con AWS WAF
- Controlli di autorizzazione integrati, con supporto per chiavi API, IAM, Amazon Cognito, provider OpenID Connect e autorizzazione Lambda per la logica personalizzata.
- Unita per supportare casi d'uso APIs federati

[Per ulteriori dettagli su ciascuna di queste funzionalità, consulta AWS AppSync Funzionalità.](#)

Sei un AWS AppSync utente alle prime armi?

Consigliamo AWS AppSync agli utenti alle prime armi di iniziare leggendo le seguenti sezioni:

- Se non conosci GraphQL, consulta la. [Guida introduttiva: creazione della prima API GraphQL](#)
- Se stai creando applicazioni che utilizzano GraphQL APIs, consulta [Creazione di un'applicazione client](#) e [the section called “Utilizzo di abbonamenti per applicazioni di dati in tempo reale”](#)
- Se stai cercando informazioni sul resolver GraphQL, consulta quanto segue:

JavaScript/TypeScript

- [Tutorial Resolver \(\) JavaScript](#)
- [Riferimento al resolver \(\) JavaScript](#)

VTL

- [Tutorial Resolver \(VTL\)](#)
- [Riferimento al modello di mappatura Resolver \(VTL\)](#)
- [Se stai cercando progetti di AWS AppSync esempio, aggiornamenti e altro, consulta il blog. AppSync](#)

Servizi correlati

Se stai creando un'app web o mobile partendo da zero, valuta la possibilità di utilizzarla [AWS Amplify](#). Amplify AWS AppSync sfrutta e AWS altri servizi per aiutarti a creare app web e mobili più solide e potenti con meno lavoro.

Prezzi per AWS AppSync

AWS AppSync ha un prezzo basato su milioni di richieste e aggiornamenti. La memorizzazione nella cache comporta un costo aggiuntivo. Per ulteriori informazioni, consulta [Prezzi di AWS AppSync](#).

Di seguito sono elencate le eccezioni ai prezzi generali AWS AppSync :

- Non viene addebitato alcun costo per le richieste per errori di autorizzazione e autenticazione.
- Non sono previsti addebiti per le chiamate ai metodi che richiedono chiavi API, in caso di chiavi API mancanti o non valide.

GraphQL e architettura AWS AppSync

Note

Questa guida presuppone che l'utente abbia una conoscenza pratica dello stile architettonico REST. Ti consigliamo di esaminare questo e altri argomenti di front-end prima di utilizzare GraphQL e AWS AppSync

GraphQL è un linguaggio di interrogazione e manipolazione per. APIs GraphQL fornisce una sintassi flessibile e intuitiva per descrivere i requisiti e le interazioni dei dati. Consente agli sviluppatori di chiedere esattamente ciò che è necessario e ottenere risultati prevedibili. Consente inoltre di accedere a più fonti in un'unica richiesta, riducendo il numero di chiamate di rete e i requisiti di larghezza di banda, risparmiando così la durata della batteria e i cicli di CPU utilizzati dalle applicazioni.

L'aggiornamento dei dati è reso semplice dalle mutazioni, che consentono agli sviluppatori di descrivere come i dati dovrebbero cambiare. GraphQL facilita anche la configurazione rapida di soluzioni in tempo reale tramite abbonamenti. Tutte queste funzionalità combinate, insieme a potenti strumenti di sviluppo, rendono GraphQL essenziale per la gestione dei dati delle applicazioni.

GraphQL è un'alternativa a REST. RESTful l'architettura è attualmente una delle soluzioni più popolari per la comunicazione client-server. È incentrato sul concetto che le tue risorse (dati) vengano esposte da un URL. Questi URLs possono essere utilizzati per accedere e manipolare i dati tramite operazioni CRUD (creazione, lettura, aggiornamento, eliminazione) sotto forma di metodi HTTP come GET, POST e DELETE. Il vantaggio di REST è che è relativamente semplice da imparare e implementare. Puoi configurarti rapidamente RESTful APIs per chiamare un'ampia gamma di servizi.

Tuttavia, la tecnologia sta diventando sempre più complicata. Man mano che le applicazioni, gli strumenti e i servizi iniziano a scalare per un pubblico mondiale, la necessità di architetture veloci e scalabili è di fondamentale importanza. REST presenta molte lacune quando si tratta di operazioni scalabili. Vedi questo [caso d'uso](#) per un esempio.

Nelle sezioni seguenti, esamineremo alcuni dei concetti correlati RESTful APIs. Presenteremo quindi GraphQL e come funziona.

Per ulteriori informazioni su GraphQL e sui vantaggi della migrazione a AWS, consulta la guida [decisionale alle implementazioni di GraphQL](#).

Argomenti

- [Che cos'è un'API?](#)
- [Che cos'è REST?](#)
- [Che cos'è GraphQL?](#)
- [Confronto tra REST e GraphQL](#)
- [Perché usare GraphQL rispetto a REST?](#)
- [Componenti di un'API GraphQL](#)
- [Proprietà aggiuntive di GraphQL](#)

Che cos'è un'API?

Un'interfaccia di programmazione delle applicazioni (API) definisce le regole da seguire per comunicare con altri sistemi software. Gli sviluppatori espongono o creano APIs in modo che altre applicazioni possano comunicare con le loro applicazioni in modo programmatico. Ad esempio, l'applicazione timesheet espone un'API che richiede il nome completo di un dipendente e un intervallo di date. Quando riceve queste informazioni, elabora internamente la scheda attività del dipendente e restituisce il numero di ore lavorate in quell'intervallo di date.

Puoi pensare a un'API Web come a un gateway tra client e risorse sul Web.

Client

I client sono utenti che desiderano accedere alle informazioni dal Web. Il client può essere una persona o un sistema software che utilizza l'API. Ad esempio, gli sviluppatori possono scrivere programmi che accedono ai dati meteorologici da un sistema meteorologico. Oppure puoi accedere agli stessi dati dal tuo browser quando visiti direttamente il sito web meteo.

Resources

Le risorse sono le informazioni che le diverse applicazioni forniscono ai propri clienti. Le risorse possono essere immagini, video, testo, numeri o qualsiasi tipo di dati. La macchina che fornisce la risorsa al client viene anche chiamata server. Le organizzazioni APIs lo utilizzano per condividere risorse e fornire servizi Web mantenendo la sicurezza, il controllo e l'autenticazione. Inoltre, APIs aiutale a determinare quali clienti possono accedere a risorse interne specifiche.

Che cos'è REST?

Ad alto livello, Representational State Transfer (REST) è un'architettura software che impone condizioni sul funzionamento di un'API. REST è stato inizialmente creato come linea guida per gestire la comunicazione su una rete complessa come Internet. È possibile utilizzare l'architettura basata su REST per supportare comunicazioni affidabili e ad alte prestazioni su larga scala. Puoi implementarlo e modificarlo facilmente, offrendo visibilità e portabilità multiplatforma a qualsiasi sistema API.

Gli sviluppatori di API possono progettare APIs utilizzando diverse architetture. APIs che seguono lo stile architettonico REST sono chiamate REST. APIs e servizi Web che implementano l'architettura REST sono chiamati servizi RESTful Web. Il termine RESTful API si riferisce generalmente al RESTful web APIs. Tuttavia, è possibile utilizzare i termini API REST e RESTful API in modo intercambiabile.

Di seguito sono riportati alcuni dei principi dello stile architettonico REST:

Interfaccia uniforme

L'interfaccia uniforme è fondamentale per la progettazione di qualsiasi RESTful servizio web. Indica che il server trasferisce le informazioni in un formato standard. La risorsa formattata è chiamata rappresentazione in REST. Questo formato può essere diverso dalla rappresentazione interna della risorsa nell'applicazione server. Ad esempio, il server può memorizzare i dati come testo ma inviarli in un formato di rappresentazione HTML.

L'interfaccia uniforme impone quattro vincoli architettonici:

1. Le richieste devono identificare le risorse. Lo fanno utilizzando un identificatore di risorsa uniforme.
2. I client dispongono di informazioni sufficienti nella rappresentazione delle risorse per modificare o eliminare la risorsa, se lo desiderano. Il server soddisfa questa condizione inviando metadati che descrivono ulteriormente la risorsa.
3. I clienti ricevono informazioni su come elaborare ulteriormente la rappresentazione. Il server ottiene ciò inviando messaggi autodescrittivi che contengono metadati su come il client può utilizzarli al meglio.
4. I clienti ricevono informazioni su tutte le altre risorse correlate di cui hanno bisogno per completare un'attività. Il server ottiene ciò inviando collegamenti ipertestuali nella rappresentazione in modo che i client possano scoprire dinamicamente più risorse.

Apolidia

Nell'architettura REST, l'apolidia si riferisce a un metodo di comunicazione in cui il server completa ogni richiesta del client indipendentemente da tutte le richieste precedenti. I client possono richiedere risorse in qualsiasi ordine e ogni richiesta è stateless o isolata dalle altre richieste. Questo vincolo di progettazione dell'API REST implica che il server possa comprendere e soddisfare completamente la richiesta ogni volta.

Sistema a strati

In un'architettura di sistema a più livelli, il client può connettersi ad altri intermediari autorizzati tra il client e il server e continuerà a ricevere risposte dal server. I server possono anche trasmettere richieste ad altri server. È possibile progettare il servizio RESTful Web in modo che venga eseguito su più server con più livelli come sicurezza, applicazione e logica aziendale, che collaborano per soddisfare le richieste dei clienti. Questi livelli rimangono invisibili al client.

Cacheabilità

RESTful i servizi web supportano la memorizzazione nella cache, che è il processo di memorizzazione di alcune risposte sul client o su un intermediario per migliorare i tempi di risposta del server. Ad esempio, supponiamo di visitare un sito Web con immagini di intestazione e piè di pagina comuni su ogni pagina. Ogni volta che visiti una nuova pagina del sito Web, il server deve inviare nuovamente le stesse immagini. Per evitare ciò, il client memorizza nella cache o memorizza queste immagini dopo la prima risposta e quindi utilizza le immagini direttamente dalla cache. RESTful i servizi web controllano la memorizzazione nella cache utilizzando risposte API che si definiscono inseribili nella cache o non memorizzabili nella cache.

Che cos'è un'API? RESTful

RESTful L'API è un'interfaccia utilizzata da due sistemi informatici per scambiare informazioni in modo sicuro su Internet. La maggior parte delle applicazioni aziendali deve comunicare con altre applicazioni interne e di terze parti per eseguire varie attività. Ad esempio, per generare buste paga mensili, il sistema contabile interno deve condividere i dati con il sistema bancario del cliente per automatizzare la fatturazione e comunicare con un'applicazione interna per la scheda attività. RESTful APIs supportano questo scambio di informazioni perché seguono standard di comunicazione software sicuri, affidabili ed efficienti.

Come RESTful APIs funziona?

La funzione di base di un' RESTful API è la stessa della navigazione in Internet. Il client contatta il server utilizzando l'API quando richiede una risorsa. Gli sviluppatori di API spiegano come il client deve utilizzare l'API REST nella documentazione dell'API dell'applicazione server. Questi sono i passaggi generali per qualsiasi chiamata all'API REST:

1. Il client invia una richiesta al server. Il client segue la documentazione dell'API per formattare la richiesta in modo comprensibile al server.
2. Il server autentica il client e conferma che il client ha il diritto di effettuare tale richiesta.
3. Il server riceve la richiesta e la elabora internamente.
4. Il server restituisce una risposta al client. La risposta contiene informazioni che indicano al client se la richiesta è andata a buon fine. La risposta include anche tutte le informazioni richieste dal client.

I dettagli della richiesta e della risposta dell'API REST variano leggermente a seconda di come gli sviluppatori dell'API progettano l'API.

Che cos'è GraphQL?

GraphQL è sia un linguaggio di query che un runtime per APIs l'esecuzione di tali query. GraphQL consente ai clienti di richiedere esattamente i dati di cui hanno bisogno, fornendo un'alternativa più flessibile ed efficiente a REST in molti scenari. A differenza di REST, che si basa su endpoint predefiniti, GraphQL utilizza un unico endpoint in cui i clienti possono specificare i propri requisiti di dati sotto forma di query e mutazioni.

Vedi [Componenti di un'API GraphQL](#) per ulteriori informazioni su come è strutturato GraphQL. APIs

Confronto tra REST e GraphQL

APIs (Application Programming Interfaces) svolgono un ruolo cruciale nel facilitare lo scambio di dati tra le applicazioni. Come affermato in precedenza, APIs sono emersi due approcci importanti per la progettazione: GraphQL e REST. Sebbene entrambi abbiano lo scopo fondamentale di abilitare la comunicazione client-server, differiscono in modo significativo nell'implementazione e nei casi d'uso.

GraphQL e REST condividono diverse caratteristiche chiave:

1. Modello client-server: entrambi utilizzano un'architettura client-server per lo scambio di dati.

2. Apolidia: nessuno dei due conserva le informazioni sulla sessione del client tra una richiesta e l'altra.
3. Basato su HTTP: entrambi utilizzano in genere HTTP come protocollo di comunicazione sottostante.
4. Progettazione orientata alle risorse: entrambi progettano lo scambio di dati attorno alle risorse, che si riferiscono a qualsiasi dato o oggetto a cui il client può accedere e manipolare tramite l'API.
5. Flessibilità del formato dei dati: JSON è il formato di scambio dati più utilizzato in entrambi, sebbene siano supportati anche altri formati come XML e HTML.
6. Indipendenti dal linguaggio e dal database: entrambi possono funzionare con qualsiasi linguaggio di programmazione o struttura di database, il che li rende altamente interoperabili.
7. Supporto per la memorizzazione nella cache: entrambi supportano la memorizzazione nella cache, consentendo a client e server di archiviare i dati a cui si accede di frequente per migliorare le prestazioni.

Pur condividendo alcuni principi fondamentali, GraphQL e REST differiscono in modo significativo nel loro approccio alla progettazione delle API e al recupero dei dati:

1. Struttura delle richieste e recupero dei dati

REST utilizza diversi metodi HTTP (GET, POST, PUT, DELETE) per eseguire operazioni sulle risorse. Ciò richiede spesso più endpoint per risorse diverse, il che può portare a inefficienze nel recupero dei dati. Ad esempio, l'esecuzione di un'operazione GET per recuperare i dati di un utente può portare a un recupero eccessivo o insufficiente dei dati. Per ottenere i dati corretti, è possibile eseguire operazioni di troncamento o più operazioni.

GraphQL utilizza un unico endpoint per tutte le operazioni. Si basa su interrogazioni per il recupero dei dati e su mutazioni per la modifica dei dati. I client possono utilizzare le query per recuperare esattamente i dati di cui hanno bisogno in un'unica richiesta, il che riduce il sovraccarico della rete riducendo al minimo il trasferimento dei dati.

2. Schema lato server

REST non richiede uno schema lato server, sebbene uno possa essere definito opzionalmente per una progettazione e una documentazione efficienti delle API.

GraphQL utilizza uno schema lato server fortemente tipizzato per definire dati e servizi dati. Lo schema, scritto in GraphQL Schema Definition Language (SDL), include tipi di oggetti e campi per ogni oggetto e funzioni resolver lato server che definiscono le operazioni per ogni campo.

3. Funzione Versioni multiple

REST spesso include il controllo delle versioni nell'URL, il che può portare al mantenimento simultaneo di più versioni dell'API. Il controllo delle versioni non è obbligatorio, ma può aiutare a prevenire modifiche irreversibili.

GraphQL promuove un'evoluzione continua dell'API senza il controllo esplicito delle versioni richiedendo la compatibilità con le versioni precedenti. I campi eliminati restituiscono messaggi di errore, mentre i tag obsoleti eliminano gradualmente i vecchi campi e restituiscono messaggi di avviso.

4. Gestione degli errori

REST è tipizzato in modo debole e richiede che la gestione degli errori sia integrata nel codice circostante. Ciò potrebbe non identificare automaticamente gli errori relativi al tipo (ad esempio, l'analisi di un numero come testo).

Al contrario, GraphQL è fortemente tipizzato e richiede una definizione dello schema completa. Ciò consente al servizio di identificare automaticamente molti errori di richiesta con un elevato livello di dettaglio.

5. Casi d'uso

REST è più adatto per:

- Applicazioni più piccole con requisiti di dati meno complessi.
- Scenari in cui i dati e le operazioni vengono utilizzati in modo simile da tutti i client.
- Applicazioni senza esigenze complesse di interrogazione dei dati.

GraphQL è più adatto per:

- Scenari con larghezza di banda limitata, in cui ridurre al minimo le richieste e le risposte è fondamentale.
- Applicazioni con più fonti di dati che devono essere combinate su un unico endpoint.
- Casi in cui le richieste dei clienti variano in modo significativo e prevedono strutture di risposta diverse.

Nota che è possibile utilizzare sia GraphQL che REST APIs all'interno di una singola applicazione per diverse aree di funzionalità. Inoltre, puoi aggiornare un' RESTful API per includere funzionalità GraphQL senza una riscrittura completa. Vedi [Come creare resolver GraphQL per sorgenti di AWS dati per un esempio.](#)

Perché usare GraphQL rispetto a REST?

REST è uno degli stili architetturici fondamentali del web. APIs Tuttavia, man mano che il mondo diventa più interconnesso, la necessità di sviluppare applicazioni robuste e scalabili diventerà una questione sempre più urgente. Sebbene REST venga spesso utilizzato per creare siti Web APIs, sono stati identificati diversi inconvenienti ricorrenti RESTful nelle implementazioni:

1. Richieste di dati: utilizzando RESTful APIs, in genere si richiedono i dati necessari tramite gli endpoint. Il problema sorge quando si hanno dati che potrebbero non essere impacchettati in modo così ordinato. I dati necessari possono essere protetti da più livelli di astrazione e l'unico modo per recuperarli è utilizzare più endpoint, il che significa effettuare più richieste per estrarre tutti i dati.
2. Overfetching e underfetching: per aggravare i problemi legati alle richieste multiple, i dati di ciascun endpoint sono definiti in modo rigoroso, il che significa che restituirai tutti i dati definiti per quell'API, anche se tecnicamente non li volevi.

Ciò può comportare un recupero eccessivo, il che significa che le nostre richieste restituiscono dati superflui. Ad esempio, supponiamo che tu stia richiedendo i dati del personale dell'azienda e desideri conoscere i nomi dei dipendenti di un determinato reparto. L'endpoint che restituisce i dati conterrà i nomi, ma potrebbe contenere anche altri dati come il titolo professionale o la data di nascita. Poiché l'API è fissa, non puoi semplicemente richiedere i nomi; il resto dei dati viene fornito con essa.

La situazione opposta, in cui non restituiamo una quantità sufficiente di dati, si chiama underfetching. Per ottenere tutti i dati richiesti, potrebbe essere necessario effettuare più richieste al servizio. A seconda di come sono strutturati i dati, potreste imbattervi in interrogazioni inefficienti con conseguenti problemi come il temuto problema $n+1$.

3. Iterazioni di sviluppo lente: molti sviluppatori le personalizzano RESTful APIs per adattarsi al flusso delle loro applicazioni. Tuttavia, man mano che le applicazioni crescono, sia il front-end che il backend potrebbero richiedere modifiche estese. Di conseguenza, APIs potrebbero non adattarsi più alla forma dei dati in modo efficiente o di impatto. Ciò si traduce in iterazioni di prodotto più lente a causa della necessità di modifiche all'API.
4. Prestazioni su larga scala: a causa di questi problemi di aggravamento, ci sono molte aree in cui la scalabilità ne risentirà. Le prestazioni sul lato applicativo potrebbero risentirne perché le richieste restituiranno troppi o troppo pochi dati (con conseguente aumento delle richieste). Entrambe le situazioni causano inutili sollecitazioni sulla rete con conseguenti scarse prestazioni. Per quanto

riguarda gli sviluppatori, la velocità di sviluppo può essere ridotta perché i dati richiesti APIs sono fissi e non sono più sufficienti.

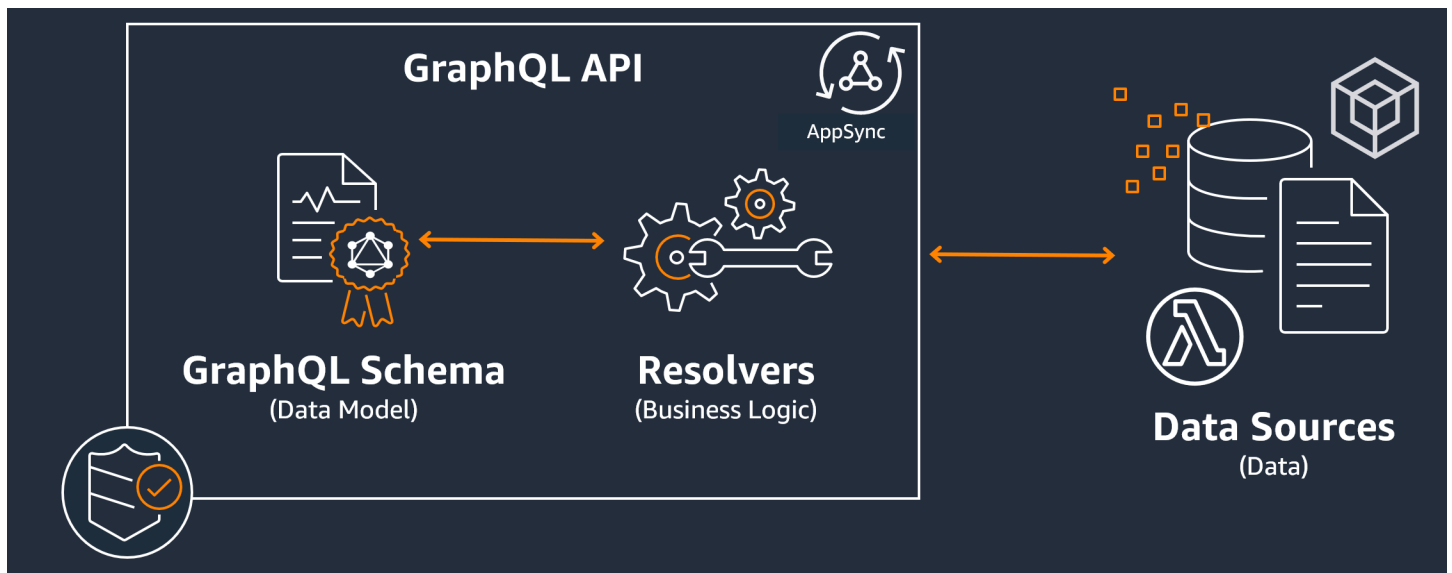
Il punto di forza di GraphQL è quello di superare gli svantaggi di REST. Ecco alcune delle soluzioni chiave che GraphQL offre agli sviluppatori:

1. **Endpoint singoli:** GraphQL utilizza un singolo endpoint per interrogare i dati. Non è necessario crearne più di uno APIs per adattarlo alla forma dei dati. Ciò si traduce in un minor numero di richieste che passano attraverso la rete.
2. **Recupero:** GraphQL risolve i problemi perenni di sovra e insufficiente recupero semplicemente definendo i dati necessari. GraphQL ti consente di modellare i dati in base alle tue esigenze in modo da ricevere solo ciò che hai richiesto.
3. **Astrazione:** APIs GraphQL contiene alcuni componenti e sistemi che descrivono i dati utilizzando uno standard indipendente dal linguaggio. In altre parole, la forma e la struttura dei dati sono standardizzate in modo che sia il front-end che il backend sappiano come verranno inviati sulla rete. Ciò consente agli sviluppatori di entrambe le parti di lavorare con i sistemi GraphQL e non con essi.
4. **Iterazioni rapide:** a causa della standardizzazione dei dati, potrebbero non essere necessarie modifiche su un lato dello sviluppo sull'altro. Ad esempio, le modifiche alla presentazione del frontend potrebbero non comportare modifiche estese al backend perché GraphQL consente di modificare facilmente le specifiche dei dati. Puoi semplicemente definire o modificare la forma dei dati per adattarla alle esigenze dell'applicazione man mano che cresce. Ciò si traduce in un minor potenziale di lavoro di sviluppo.

Questi sono solo alcuni dei vantaggi di GraphQL. Nelle prossime sezioni, scoprirai come è strutturato GraphQL e le proprietà che lo rendono un'alternativa unica a REST.

Componenti di un'API GraphQL

Un'API GraphQL standard è composta da un unico schema che gestisce la forma dei dati che verranno interrogati. Lo schema è collegato a una o più fonti di dati come un database o una funzione Lambda. Tra i due si trovano uno o più resolver che gestiscono la logica aziendale per le tue richieste. Ogni componente svolge un ruolo importante nell'implementazione di GraphQL. Le sezioni seguenti introdurranno questi tre componenti e il ruolo che svolgono nel servizio GraphQL.



Argomenti

- [Schemi GraphQL](#)
- [Fonti di dati](#)
- [Risolutori](#)

Schemi GraphQL

Lo schema GraphQL è alla base di un'API GraphQL. Serve come modello che definisce la forma dei dati. È anche un contratto tra il client e il server che definisce come i dati verranno recuperati and/or e modificati.

Gli schemi GraphQL sono scritti nello Schema Definition Language (SDL). SDL è composto da tipi e campi con una struttura consolidata:

- **Tipi:** I tipi sono il modo in cui GraphQL definisce la forma e il comportamento dei dati. GraphQL supporta una moltitudine di tipi che verranno spiegati più avanti in questa sezione. Ogni tipo definito nello schema conterrà il proprio ambito. All'interno dell'ambito ci saranno uno o più campi che possono contenere un valore o una logica che verrà utilizzata nel servizio GraphQL. I tipi ricoprono molti ruoli diversi, i più comuni sono gli oggetti o gli scalari (tipi di valori primitivi).
- **Campi:** i campi esistono nell'ambito di un tipo e contengono il valore richiesto dal servizio GraphQL. Sono molto simili alle variabili di altri linguaggi di programmazione. La forma dei dati definiti nei campi determinerà il modo in cui i dati sono strutturati in un' request/response operazione. Ciò

consente agli sviluppatori di prevedere cosa verrà restituito senza sapere come viene implementato il backend del servizio.

Per visualizzare l'aspetto di uno schema, esaminiamo il contenuto di un semplice schema GraphQL. Nel codice di produzione, lo schema si trova in genere in un file chiamato `schema.graphql` o `schema.json`. Supponiamo che stiamo esaminando un progetto che implementa un servizio GraphQL. Questo progetto archivia i dati del personale dell'azienda e il `schema.graphql` file viene utilizzato per recuperare i dati sul personale e aggiungere nuovo personale a un database. Il codice potrebbe essere simile al seguente:

`schema.graphql`

```
type Person {
  id: ID!
  name: String
  age: Int
}
type Query {
  people: [Person]
}
type Mutation {
  addPerson(id: ID!, name: String, age: Int): Person
}
```

Possiamo vedere che ci sono tre tipi definiti nello schema: `Person`, `Query`, e `Mutation`.

Guardando `Person`, possiamo immaginare che questo sia il modello per un esempio di un dipendente dell'azienda, il che renderebbe questo tipo un oggetto. All'interno del suo ambito `id`, `name`, vediamo `age`. Questi sono i campi che definiscono le proprietà di `Person`. Ciò significa che la nostra fonte `Person` di dati memorizza ciascuno `name` come tipo `String` scalare (primitivo) e `age` come tipo `Int` scalare (primitivo). `id` funziona come un identificatore speciale e univoco per ciascuno. `Person` è anche un valore obbligatorio, come indicato dal `!` simbolo.

I due tipi di oggetti successivi si comportano in modo diverso. GraphQL riserva alcune parole chiave per tipi di oggetti speciali che definiscono il modo in cui i dati verranno popolati nello schema. Un `Query` tipo recupererà i dati dalla fonte. Nel nostro esempio, la nostra query potrebbe recuperare `Person` oggetti da un database. Questo potrebbe ricordarvi le GET operazioni RESTful terminologiche. A `Mutation` modificherà i dati. Nel nostro esempio, la nostra mutazione può aggiungere altri `Person` oggetti al database. Questo potrebbe ricordarti di operazioni che cambiano

lo stato come `oPUT`. `POST` I comportamenti di tutti i tipi di oggetti speciali verranno spiegati più avanti in questa sezione.

Supponiamo che `Query` nel nostro esempio recuperi qualcosa dal database. Se osserviamo i campi di `Query`, vediamo un campo chiamato `people`. Il suo valore di campo è `[Person]`. Ciò significa che vogliamo recuperare alcune istanze `Person` del database. Tuttavia, l'aggiunta di parentesi indica che vogliamo restituire un elenco di tutte le `Person` istanze e non solo uno specifico.

Il `Mutation` tipo è responsabile dell'esecuzione di operazioni di modifica dello stato come la modifica dei dati. Una mutazione è responsabile dell'esecuzione di alcune operazioni di modifica dello stato sulla fonte di dati. Nel nostro esempio, la nostra mutazione contiene un'operazione chiamata `addPerson` che aggiunge un nuovo `Person` oggetto al database. La mutazione utilizza a `Person` e prevede un input per i campi `idname`, e. `age`

A questo punto, forse vi starete chiedendo come `addPerson` funzionano operazioni come quelle senza un'implementazione di codice, dato che presumibilmente esegue un certo comportamento e assomiglia molto a una funzione con un nome di funzione e parametri. Attualmente non funzionerà perché uno schema funge solo da dichiarazione. Per implementare il comportamento di `addPerson`, dovremmo aggiungervi un resolver. Un resolver è un'unità di codice che viene eseguita ogni volta che viene chiamato il campo associato (in questo caso, l'`addPerson` operazione). Se desideri utilizzare un'operazione, a un certo punto dovrai aggiungere l'implementazione del resolver. In un certo senso, puoi pensare all'operazione dello schema come alla dichiarazione di funzione e al resolver come alla definizione. I resolver verranno spiegati in una sezione diversa.

Questo esempio mostra solo i modi più semplici in cui uno schema può manipolare i dati. Puoi creare applicazioni complesse, robuste e scalabili sfruttando le funzionalità di GraphQL e. AWS AppSync. Nella prossima sezione, definiremo tutti i diversi tipi e comportamenti sul campo che puoi utilizzare nel tuo schema.

Tipi GraphQL

GraphQL supporta molti tipi diversi. Come hai visto nella sezione precedente, i tipi definiscono la forma o il comportamento dei dati. Sono gli elementi costitutivi fondamentali di uno schema GraphQL.

I tipi possono essere classificati in input e output. Gli input sono tipi che possono essere passati come argomento per i tipi di oggetti speciali (`Query`, ecc.) `Mutation`, mentre i tipi di output vengono utilizzati strettamente per archiviare e restituire dati. Di seguito è riportato un elenco di tipi e delle relative categorizzazioni:

- **Oggetti:** un oggetto contiene campi che descrivono un'entità. Ad esempio, un oggetto potrebbe essere qualcosa come un oggetto `book` con campi che ne descrivono le caratteristiche come `authorName` `publishingYear`, ecc. Sono strettamente tipi di output.
- **Scalari:** sono tipi primitivi come `int`, `string`, ecc. In genere vengono assegnati ai campi. Usando il `authorName` campo come esempio, potrebbe essere assegnato lo `String` scalare per memorizzare un nome come «John Smith». Gli scalari possono essere sia di tipo di input che di output.
- **Ingressi:** gli input consentono di passare un gruppo di campi come argomento. Sono strutturati in modo molto simile agli oggetti, ma possono essere passati come argomenti a oggetti speciali. Gli input consentono di definire scalari, enumerazioni e altri input nel relativo ambito. Gli input possono essere solo tipi di input.
- **Oggetti speciali:** gli oggetti speciali eseguono operazioni di modifica dello stato e svolgono la maggior parte del lavoro pesante del servizio. Esistono tre tipi di oggetti speciali: interrogazione, mutazione e sottoscrizione. Le query in genere recuperano i dati; le mutazioni manipolano i dati; le sottoscrizioni si aprono e mantengono una connessione bidirezionale tra client e server per una comunicazione costante. Gli oggetti speciali non vengono né input né output date le loro funzionalità.
- **Enumerazioni:** le enumerazioni sono elenchi predefiniti di valori legali. Se chiami un enum, i suoi valori possono essere solo quelli definiti nel suo ambito. Ad esempio, se aveste un enum chiamato `trafficLights` rappresenta un elenco di segnali stradali, potrebbe avere valori come `redLight` e `greenLight` ma no. `purpleLight` Un vero semaforo avrà solo un certo numero di segnali, quindi puoi usare l'enum per definirli e forzarli a essere gli unici valori legali durante il riferimento. `trafficLight` Gli enum possono essere sia di tipo di input che di output.
- **Unioni/interfacce:** le unioni consentono di restituire uno o più elementi in una richiesta a seconda dei dati richiesti dal client. Ad esempio, se si dispone `Book` di un tipo con un `title` campo e un `Author` tipo con un `name` campo, è possibile creare un'unione tra entrambi i tipi. Se il cliente volesse cercare in un database la frase «Giulio Cesare», il sindacato potrebbe restituire Giulio Cesare (l'opera di William Shakespeare) tratto da *Book title* e Giulio Cesare (l'autore di *Commentarii de Bello Gallico*) dal. `Author name` Le unioni possono essere solo tipi di output.

Le interfacce sono insiemi di campi che gli oggetti devono implementare. È un po' simile alle interfacce nei linguaggi di programmazione come Java, in cui è necessario implementare i campi definiti nell'interfaccia. Ad esempio, supponiamo che tu abbia creato un'interfaccia chiamata `Book` che contenesse un `title` campo. Supponiamo che in seguito tu `Novel` abbia creato un tipo chiamato `implementatoBook`. `Novel` Dovresti includere un `title` campo. Tuttavia, `Novel` potresti

includere anche altri campi non presenti nell'interfaccia come `pageCount` o `ISBN`. Le interfacce possono essere solo tipi di output.

Le sezioni seguenti spiegheranno come funziona ogni tipo in GraphQL.

Oggetti

Gli oggetti GraphQL sono il tipo principale che vedrai nel codice di produzione. In GraphQL, puoi pensare a un oggetto come a un raggruppamento di campi diversi (simili alle variabili in altri linguaggi), con ogni campo definito da un tipo (tipicamente uno scalare o un altro oggetto) che può contenere un valore. Gli oggetti rappresentano un'unità di dati che può retrieved/manipulated provenire dall'implementazione del servizio.

I tipi di oggetti vengono dichiarati utilizzando la `Type` parola chiave. Modifichiamo leggermente il nostro esempio di schema:

```
type Person {  
  id: ID!  
  name: String  
  age: Int  
  occupation: Occupation  
}  
  
type Occupation {  
  title: String  
}
```

I tipi di oggetti qui sono `Person` e `Occupation`. Ogni oggetto ha i propri campi con i propri tipi. Una caratteristica di GraphQL è la possibilità di impostare campi su altri tipi. Puoi vedere che il `occupation` campo `Person` contiene un tipo di `Occupation` oggetto. Possiamo fare questa associazione perché GraphQL descrive solo i dati e non l'implementazione del servizio.

Scalari

Gli scalari sono essenzialmente tipi primitivi che contengono valori. Nel AWS AppSync, esistono due tipi di scalari: gli scalari e gli AWS AppSync scalari GraphQL predefiniti. Gli scalari vengono in genere utilizzati per memorizzare i valori dei campi all'interno dei tipi di oggetti. I tipi GraphQL predefiniti includono `Int`, `Float`, `String`, `Boolean`, e `ID`. Usiamo nuovamente l'esempio precedente:

```
type Person {
```

```
id: ID!  
name: String  
age: Int  
occupation: Occupation  
}  
  
type Occupation {  
  title: String  
}
```

Individuando i `title` campi `name` e, entrambi contengono uno `String` scalare. `Name` potrebbe restituire un valore di stringa come "John Smith" e il titolo potrebbe restituire qualcosa come "»firefighter. Alcune implementazioni GraphQL supportano anche scalari personalizzati che utilizzano la `Scalar` parola chiave e implementano il comportamento del tipo. Tuttavia, AWS AppSync attualmente non supporta scalari personalizzati. Per un elenco di scalari, vedi Tipi [scalari](#) in. AWS AppSync

Input

A causa del concetto di tipi di input e output, esistono alcune restrizioni quando si passano argomenti. I tipi che di solito devono essere passati, in particolare gli oggetti, sono limitati. È possibile utilizzare il tipo di input per aggirare questa regola. Gli input sono tipi che contengono scalari, enumerazioni e altri tipi di input.

Gli input sono definiti utilizzando la parola chiave: `input`

```
type Person {  
  id: ID!  
  name: String  
  age: Int  
  occupation: Occupation  
}  
  
type Occupation {  
  title: String  
}  
  
input personInput {  
  id: ID!  
  name: String  
  age: Int  
  occupation: occupationInput
```

```
}

input occupationInput {
  title: String
}
```

Come puoi vedere, possiamo avere input separati che imitano il tipo originale. Questi input verranno spesso utilizzati nelle operazioni sul campo in questo modo:

```
type Person {
  id: ID!
  name: String
  age: Int
  occupation: Occupation
}

type Occupation {
  title: String
}

input occupationInput {
  title: String
}

type Mutation {
  addPerson(id: ID!, name: String, age: Int, occupation: occupationInput): Person
}
```

Nota come stiamo ancora passando `occupationInput` al posto di `Occupation` creare un `Person`.

Questo è solo uno scenario per gli input. Non hanno necessariamente bisogno di copiare gli oggetti in scala 1:1e, nel codice di produzione, molto probabilmente non lo userete in questo modo. È buona norma sfruttare gli schemi GraphQL definendo solo ciò che è necessario inserire come argomenti.

Inoltre, gli stessi input possono essere utilizzati in più operazioni, ma non è consigliabile farlo. Ogni operazione dovrebbe idealmente contenere una propria copia unica degli input nel caso in cui i requisiti dello schema cambino.

Oggetti speciali

GraphQL riserva alcune parole chiave per oggetti speciali che definiscono alcune delle logiche aziendali relative al modo in cui lo schema utilizzerà retrieve/manipulate i dati. Al massimo, può

esserci una di queste parole chiave in uno schema. Fungono da punti di ingresso per tutti i dati richiesti che i tuoi clienti eseguono sul tuo servizio GraphQL.

Gli oggetti speciali vengono definiti anche utilizzando la `type` parola chiave. Sebbene vengano utilizzati in modo diverso dai normali tipi di oggetti, la loro implementazione è molto simile.

Queries

Le query sono molto simili alle GET operazioni in quanto eseguono un recupero di sola lettura per ottenere dati dalla fonte. In GraphQL, Query definisce tutti i punti di ingresso per i client che effettuano richieste verso il tuo server. Ci sarà sempre un'implementazione GraphQL Query nella tua implementazione GraphQL.

Ecco i Query tipi di oggetti modificati che abbiamo usato nel nostro precedente esempio di schema:

```
type Person {
  id: ID!
  name: String
  age: Int
  occupation: Occupation
}
type Occupation {
  title: String
}
type Query {
  people: [Person]
}
```

Il nostro Query contiene un campo chiamato `people` che restituisce un elenco di `Person` istanze dalla fonte di dati. Supponiamo di dover modificare il comportamento della nostra applicazione e ora dobbiamo restituire un elenco delle sole `Occupation` istanze per uno scopo separato. Potremmo semplicemente aggiungerlo alla query:

```
type Query {
  people: [Person]
  occupations: [Occupation]
}
```

In GraphQL, possiamo trattare la nostra query come l'unica fonte di richieste. Come puoi vedere, questo è potenzialmente molto più semplice RESTful delle implementazioni che potrebbero

utilizzare endpoint diversi per ottenere lo stesso risultato (e). `.../api/1/people` `.../api/1/occupations`

Supponendo di avere un'implementazione del resolver per questa query, ora possiamo eseguire una query vera e propria. Sebbene il Query tipo esista, dobbiamo chiamarlo esplicitamente affinché venga eseguito nel codice dell'applicazione. Questo può essere fatto usando la query parola chiave:

```
query getItems {  
  people {  
    name  
  }  
  occupations {  
    title  
  }  
}
```

Come puoi vedere, questa query viene chiamata `getItems` e restituisce `people` (un elenco di `Person` oggetti) e `occupations` (un elenco di `Occupation` oggetti). Nel `people`, stiamo restituendo solo il `name` campo di ciascuno `Person`, mentre stiamo restituendo il `title` campo di ciascuno `Occupation`. La risposta potrebbe essere simile a questa:

```
{  
  "data": {  
    "people": [  
      {  
        "name": "John Smith"  
      },  
      {  
        "name": "Andrew Miller"  
      },  
      .  
      .  
      .  
    ],  
    "occupations": [  
      {  
        "title": "Firefighter"  
      },  
      {  
        "title": "Bookkeeper"  
      },  
    ],  
  }  
}
```

```

      .
      .
      .
    ]
  }
}

```

La risposta di esempio mostra come i dati seguono la forma della query. Ogni voce recuperata viene elencata nell'ambito del campo. `people` e `occupations` stanno restituendo le cose come elenchi separati. Sebbene utile, potrebbe essere più comodo modificare la query per restituire un elenco di nomi e occupazioni delle persone:

```

query.getItems {
  people {
    name
    occupation {
      title
    }
  }
}

```

Questa è una modifica legale perché il nostro `Person` tipo contiene un `occupation` campo di tipo `Occupation`. Se elencati nell'ambito di `people`, restituiamo ciascuno `Person` di essi `name` insieme a quello associato `Occupation` da `title`. La risposta potrebbe essere simile a questa:

```

}
"data": {
  "people": [
    {
      "name": "John Smith",
      "occupation": {
        "title": "Firefighter"
      }
    },
    {
      "name": "Andrew Miller",
      "occupation": {
        "title": "Bookkeeper"
      }
    },
    .
    .
    .
  ]
}

```

```
    ]
  }
}
```

Mutations

Le mutazioni sono simili a operazioni che cambiano lo stato come `PUT`. `POST` Eseguono un'operazione di scrittura per modificare i dati nell'origine, quindi recuperano la risposta. Definiscono i punti di ingresso per le richieste di modifica dei dati. A differenza delle query, una mutazione può essere inclusa o meno nello schema a seconda delle esigenze del progetto. Ecco la mutazione dall'esempio dello schema:

```
type Mutation {
  addPerson(id: ID!, name: String, age: Int): Person
}
```

Il `addPerson` campo rappresenta un punto di ingresso che aggiunge un `Person` all'origine dati. `addPerson` è il nome del campo; `id`, `name`, e `age` sono i parametri; ed `Person` è il tipo restituito. Guardando indietro al `Person` tipo:

```
type Person {
  id: ID!
  name: String
  age: Int
  occupation: Occupation
}
```

Abbiamo aggiunto il `occupation` campo. Tuttavia, non possiamo impostare questo campo su `Occupation` direttamente perché gli oggetti non possono essere passati come argomenti; sono strettamente tipi di output. Dovremmo invece passare un input con gli stessi campi di un argomento:

```
input occupationInput {
  title: String
}
```

Possiamo anche aggiornare facilmente il nostro `addPerson` per includerlo come parametro quando creiamo nuove `Person` istanze:

```
type Mutation {
```

```
    addPerson(id: ID!, name: String, age: Int, occupation: occupationInput): Person
  }
```

Ecco lo schema aggiornato:

```
type Person {
  id: ID!
  name: String
  age: Int
  occupation: Occupation
}

type Occupation {
  title: String
}

input occupationInput {
  title: String
}

type Mutation {
  addPerson(id: ID!, name: String, age: Int, occupation: occupationInput): Person
}
```

Nota che `occupation` passerà nel `title` campo da `occupationInput` per completare la creazione dell'oggetto `Person` anziché dell'`Occupation` oggetto originale. Supponendo di avere un'implementazione del resolver per `addPerson`, ora possiamo eseguire una mutazione effettiva. Sebbene il `Mutation` tipo esista, dobbiamo chiamarlo esplicitamente affinché venga eseguito nel codice dell'applicazione. Questo può essere fatto usando la `mutation` parola chiave:

```
mutation createPerson {
  addPerson(id: ID!, name: String, age: Int, occupation: occupationInput) {
    name
    age
    occupation {
      title
    }
  }
}
```

Questa mutazione si chiama `createPerson`, ed `addPerson` è l'operazione. Per crearne una nuova `Person`, possiamo inserire gli argomenti `id`, `name`, `age`, `occupation`. Nell'ambito di `addPerson`, possiamo vedere anche altri campi come `name`, `age`, ecc. Questa è la tua risposta; questi sono i campi che verranno restituiti al termine dell'operazione `addPerson`. Ecco la parte finale dell'esempio:

```
mutation createPerson {  
  addPerson(id: "1", name: "Steve Powers", age: "50", occupation: "Miner") {  
    id  
    name  
    age  
    occupation {  
      title  
    }  
  }  
}
```

Utilizzando questa mutazione, un risultato potrebbe essere simile al seguente:

```
{  
  "data": {  
    "addPerson": {  
      "id": "1",  
      "name": "Steve Powers",  
      "age": "50",  
      "occupation": {  
        "title": "Miner"  
      }  
    }  
  }  
}
```

Come puoi vedere, la risposta ha restituito i valori richiesti nello stesso formato definito nella nostra mutazione. È buona norma restituire tutti i valori che sono stati modificati per ridurre la confusione e la necessità di ulteriori query in futuro. Le mutazioni consentono di includere più operazioni nel suo ambito. Verranno eseguite in sequenza nell'ordine indicato nella mutazione. Ad esempio, se creiamo un'altra operazione chiamata `addOccupation` che aggiunge titoli di lavoro all'origine dati, possiamo richiamarla nella mutazione successiva. `addPerson` verrà gestito per primo seguito da `addOccupation`.

Subscriptions

Gli abbonamenti vengono utilizzati [WebSockets](#) per aprire una connessione bidirezionale duratura tra il server e i suoi client. In genere, un client si iscrive o ascolta il server. Ogni volta che il server apporta una modifica sul lato server o esegue un evento, il client sottoscritto riceverà gli aggiornamenti. Questo tipo di protocollo è utile quando sono sottoscritti più client e devono essere avvisati delle modifiche che avvengono nel server o in altri client. Ad esempio, gli abbonamenti possono essere utilizzati per aggiornare i feed dei social media. Potrebbero esserci due utenti, l'utente A e l'utente B, entrambi abbonati agli aggiornamenti automatici delle notifiche ogni volta che ricevono messaggi diretti. L'utente A sul client A potrebbe inviare un messaggio diretto all'utente B sul client B. Il client dell'utente A invierebbe il messaggio diretto, che verrebbe elaborato dal server. Il server invierebbe quindi il messaggio diretto all'account dell'utente B inviando una notifica automatica al client B.

Ecco un esempio di a Subscription che potremmo aggiungere all'esempio dello schema:

```
type Subscription {  
  personAdded: Person  
}
```

Il `personAdded` campo invierà un messaggio ai clienti abbonati ogni volta che ne `Person` viene aggiunto uno nuovo alla fonte di dati. Supponendo di avere un'implementazione del resolver `perpersonAdded`, ora possiamo usare l'abbonamento. Sebbene il `Subscription` tipo esista, dobbiamo chiamarlo esplicitamente affinché venga eseguito nel codice dell'applicazione. Questo può essere fatto usando la `subscription` parola chiave:

```
subscription personAddedOperation {  
  personAdded {  
    id  
    name  
  }  
}
```

L'abbonamento viene chiamato `personAddedOperation` e l'operazione è `personAdded`. `personAdded` restituirà i campi `id` e `name` delle nuove `Person` istanze. Guardando l'esempio di mutazione, abbiamo aggiunto un'operazione che `Person` utilizza questa:

```
addPerson(id: "1", name: "Steve Powers", age: "50", occupation: "Miner")
```

Se i nostri clienti erano abbonati agli aggiornamenti di quelli appena aggiunti `Person`, potrebbero vederlo dopo l'`addPerson` esecuzione:

```
{
  "data": {
    "personAdded": {
      "id": "1",
      "name": "Steve Powers"
    }
  }
}
```

Di seguito è riportato un riepilogo di ciò che offrono gli abbonamenti:

Gli abbonamenti sono canali bidirezionali che consentono al client e al server di ricevere aggiornamenti rapidi ma costanti. In genere utilizzano il WebSocket protocollo, che crea connessioni standardizzate e sicure.

Gli abbonamenti sono agili in quanto riducono il sovraccarico di configurazione della connessione. Una volta sottoscritto, un cliente può continuare a utilizzare tale abbonamento per lunghi periodi di tempo. In genere utilizzano le risorse informatiche in modo efficiente, consentendo agli sviluppatori di personalizzare la durata dell'abbonamento e di configurare le informazioni richieste.

In generale, gli abbonamenti consentono al cliente di effettuare più abbonamenti contemporaneamente. Per quanto riguarda AWS AppSync, gli abbonamenti vengono utilizzati solo per ricevere aggiornamenti in tempo reale dal servizio. AWS AppSync Non possono essere utilizzati per eseguire interrogazioni o mutazioni.

L'alternativa principale agli abbonamenti è il polling, che invia interrogazioni a intervalli prestabiliti per richiedere dati. Questo processo è in genere meno efficiente degli abbonamenti e mette a dura prova sia il client che il backend.

Una cosa che non è stata menzionata nel nostro esempio di schema è il fatto che anche i tipi di oggetti speciali devono essere definiti in una schema radice. Quindi, quando esporti uno schema in AWS AppSync, potrebbe assomigliare a questo:

schema.graphql

```
schema {
```

```
  query: Query
  mutation: Mutation
  subscription: Subscription
}

.
.
.

type Query {
  # code goes here
}
type Mutation {
  # code goes here
}
type Subscription {
  # code goes here
}
```

Enumerazioni

Le enumerazioni, o enumerazioni, sono scalari speciali che limitano gli argomenti legali di un tipo o di un campo. Ciò significa che ogni volta che un enum viene definito nello schema, il tipo o il campo associato sarà limitato ai valori nell'enum. Gli enum sono serializzati come stringhe scalari. Nota che diversi linguaggi di programmazione possono gestire le enumerazioni GraphQL in modo diverso. Ad esempio, non JavaScript ha un supporto enum nativo, quindi i valori enum possono essere mappati invece su valori int.

Le enumerazioni vengono definite utilizzando la parola chiave. `enum` Ecco un esempio:

```
enum trafficSignals {
  solidRed
  solidYellow
  solidGreen
  greenArrowLeft
  ...
}
```

Quando si chiama `trafficLight`enum, gli argomenti possono essere `solidRed`, `solidYellow``solidGreen`, ecc. È comune usare le enumerazioni per rappresentare cose che hanno un numero distinto ma limitato di scelte.

Unioni/interfacce

Vedi [Interfacce e unioni in GraphQL](#).

Campi GraphQL

I campi rientrano nell'ambito di un tipo e contengono il valore richiesto dal servizio GraphQL. Sono molto simili alle variabili di altri linguaggi di programmazione. Ad esempio, ecco un tipo di `Person` oggetto:

```
type Person {  
  name: String  
  age: Int  
}
```

I campi in questo caso sono `name` e `age` e contengono rispettivamente un `String` e un `Int` valore. I campi oggetto come quelli mostrati sopra possono essere usati come input nei campi (operazioni) delle query e delle mutazioni. Ad esempio, vedi quanto segue: Query

```
type Query {  
  people: [Person]  
}
```

Il `people` campo richiede tutte le istanze di `Person` dalla fonte di dati. Quando aggiungi o recuperi un file `Person` nel tuo server GraphQL, puoi aspettarti che i dati seguano il formato dei tuoi tipi e campi, ovvero la struttura dei tuoi dati nello schema determina come saranno strutturati nella tua risposta:

```
}  
"data": {  
  "people": [  
    {  
      "name": "John Smith",  
      "age": "50"  
    },  
    {  
      "name": "Andrew Miller",  
      "age": "60"  
    },  
  ],  
}
```

```

    .
  .
]
}
}

```

I campi svolgono un ruolo importante nella strutturazione dei dati. Di seguito sono illustrate un paio di proprietà aggiuntive che possono essere applicate ai campi per una maggiore personalizzazione.

Elenchi

Gli elenchi restituiscono tutti gli elementi di un tipo specificato. È possibile aggiungere un elenco al tipo di campo utilizzando le parentesi[]:

```

type Person {
  name: String
  age: Int
}
type Query {
  people: [Person]
}

```

InQuery, le parentesi che lo circondano Person indicano che si desidera restituire tutte le istanze di Person dalla fonte di dati come matrice. Nella risposta, i age valori name e di ciascuno Person verranno restituiti come un unico elenco delimitato:

```

}
"data": {
  "people": [
    {
      "name": "John Smith",      # Data of Person 1
      "age": "50"
    },
    {
      "name": "Andrew Miller",  # Data of Person 2
      "age": "60"
    },
    .                            # Data of Person N
    .
    .
  ]
}

```

```
}
```

Non sei limitato a tipi di oggetti speciali. È inoltre possibile utilizzare elenchi nei campi dei tipi di oggetti normali.

Non-null

I valori non nulli indicano un campo che non può essere nullo nella risposta. È possibile impostare un campo su un valore diverso da nullo utilizzando il simbolo: !

```
type Person {  
  name: String!  
  age: Int  
}  
type Query {  
  people: [Person]  
}
```

Il nome campo non può essere esplicitamente nullo. Se dovessi interrogare l'origine dati e fornissi un input nullo per questo campo, verrebbe generato un errore.

È possibile combinare elenchi e valori non nulli. Confronta queste domande:

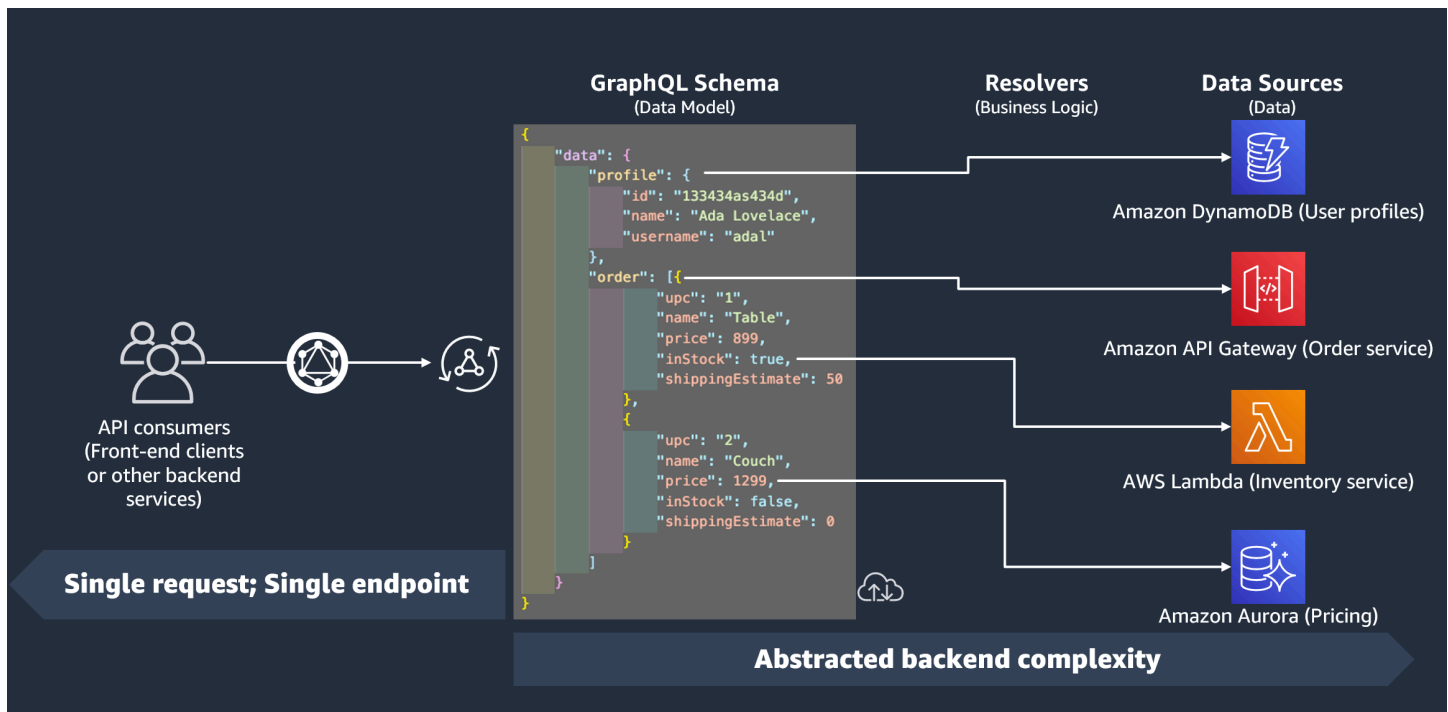
```
type Query {  
  people: [Person!]      # Use case 1  
}  
  
.  
.  
.  
  
type Query {  
  people: [Person]!      # Use case 2  
}  
  
.  
.  
.  
  
type Query {  
  people: [Person!]!      # Use case 3  
}
```

Nel caso d'uso 1, l'elenco non può contenere elementi nulli. Nel caso d'uso 2, l'elenco stesso non può essere impostato su null. Nel caso d'uso 3, l'elenco e i relativi elementi non possono essere nulli. Tuttavia, in ogni caso, è comunque possibile restituire elenchi vuoti.

Come puoi vedere, ci sono molti componenti mobili in GraphQL. In questa sezione, abbiamo mostrato la struttura di uno schema semplice e i diversi tipi e campi supportati da uno schema. Nella sezione seguente, scoprirai gli altri componenti di un'API GraphQL e come funzionano con lo schema.

Fonti di dati

Nella sezione precedente, abbiamo appreso che uno schema definisce la forma dei dati. Tuttavia, non abbiamo mai spiegato da dove provenissero quei dati. Nei progetti reali, lo schema è come un gateway che gestisce tutte le richieste fatte al server. Quando viene effettuata una richiesta, lo schema funge da singolo endpoint che si interfaccia con il client. Lo schema accederà, elaborerà e inoltrerà i dati dalla fonte dati al client. Guarda l'infografica qui sotto:



AWS AppSync e GraphQL implementano in modo eccellente le soluzioni Backend For Frontend (BFF). Lavorano in tandem per ridurre la complessità su larga scala astruendo il backend. Se il tuo servizio utilizza diverse fonti di dati e/o microservizi, puoi essenzialmente eliminare parte della complessità definendo la forma dei dati di ciascuna fonte (sottografo) in un unico schema (supergrafo). Ciò significa che l'API GraphQL non si limita a utilizzare un'unica fonte di dati. Puoi associare un numero qualsiasi di fonti di dati all'API GraphQL e specificare nel codice come interagiranno con il servizio.

Come puoi vedere nell'infografica, lo schema GraphQL contiene tutte le informazioni di cui i client hanno bisogno per richiedere dati. Ciò significa che tutto può essere elaborato in un'unica richiesta anziché in più richieste come nel caso di REST. Queste richieste passano attraverso lo schema, che è l'unico endpoint del servizio. Quando le richieste vengono elaborate, un resolver (spiegato nella sezione successiva) esegue il proprio codice per elaborare i dati dalla fonte di dati pertinente. Quando viene restituita la risposta, il sottografo collegato all'origine dati verrà popolato con i dati dello schema.

AWS AppSync supporta molti tipi di fonti di dati diversi. Nella tabella seguente, descriveremo ogni tipo, elencheremo alcuni dei vantaggi di ciascuno e forniremo link utili per un contesto aggiuntivo.

Origine dati	Description	Vantaggi	Informazioni supplementari
Amazon DynamoDB	«Amazon DynamoDB è un servizio di database NoSQL completamente gestito che offre prestazioni veloci e prevedibili con una scalabilità perfetta. DynamoDB consente di scaricare gli oneri di gestione e dimensionamento di un database distribuito in modo da non doversi più preoccupare di provisioning dell'hardware, installazione e configurazione, replica, applicazione di patch al software e dimensionamento del cluster.	• Prestazioni su larga scala: DynamoDB è progettato per garantire prestazioni costanti su qualsiasi scala. Ciò è possibile tramite l'uso di partizioni. DynamoDB partizionerà automaticamente le tabelle in diverse allocazioni che verranno archiviate e in più nodi. SSDs. Questo generalmente aumenterà la velocità di trasmissione della rete e ridurrà la latenza. • Capacità su larga scala: DynamoDB monitora il traffico e	• Documentazione ufficiale di DynamoDB • Partizioni • Dimensionamento automatico • Tolleranza agli errori • Monitoraggio • Sicurezza • GraphQL e DynamoDB • Operazioni Resolver per DynamoDB • Modello di prezzo

Origine dati	Description	Vantaggi	Informazioni supplementari
	DynamoDB offre anche la crittografia a riposo, che elimina l'onere operativo e la complessità associati alla protezione dei dati sensibili».	consente di scalare automaticamente il throughput se la rete rimane sovraccarica per lunghi periodi di tempo. - Disponibilità e tolleranza agli errori: DynamoDB è supportato da diverse regioni fisicamente isolate, ognuna contenente diverse zone di disponibilità fisicamente isolate. DynamoDB passerà automaticamente a una zona di backup in caso di interruzione del servizio. È inoltre possibile eseguire il backup e la replica dei dati manualmente per garantire la sicurezza dei dati. - Registrazione e monitoraggio: DynamoDB fornisce diversi strumenti analitici per le	

Origine dati	Description	Vantaggi	Informazioni supplementari
		tabelle. Puoi monitorare le prestazioni della tua tabella e creare allarmi per avvisarti di cambiamenti drastici al servizio. • Sicurezza: DynamoDB segue protocolli rigorosi per garantire che i dati siano conformi ai requisiti di sicurezza dell'organizzazione. • Integrazione con AWS AppSync: DynamoDB si integra perfettamente con il nostro servizio. Puoi creare nuove tabelle DynamoDB e generare automaticamente uno schema a partire da esse per semplificare il processo di sviluppo. Forniamo anche un'intera raccolta di operazioni	

Origine dati	Description	Vantaggi	Informazioni supplementari
		per richiedere facilmente dati dalle tabelle DynamoDB esistenti nel tuo account nel tuo resolver.	

Origine dati	Description	Vantaggi	Informazioni supplementari
AWS Lambda	"AWS Lambda è un servizio di elaborazione che consente di eseguire codice senza effettuare il provisioning o la gestione di server. Lambda esegue il codice su un'infrastruttura di elaborazione ad alta disponibilità e gestisce tutta l'amministrazione delle risorse di elaborazione, compresa la manutenzione del server e del sistema operativo, il provisioning e la scalabilità automatica della capacità e la registrazione. Con Lambda, tutto ciò che devi fare è fornire il codice in uno dei runtime linguistici supportati da Lambda».	• Pay-as-you-use modello: Lambda ti addebita solo quando utilizzi le sue risorse. Consentono inoltre di scalare la quantità di risorse utilizzate in base alle esigenze dell'applicazione. • Scalabilità automatica: a volte l'applicazione può richiedere una potenza di calcolo aggiuntiva per un particolare processo. Lambda ti consente di scalare automaticamente le risorse di elaborazione per adattarle alle esigenze della tua applicazione. • Tempi di implementazione più rapidi: puoi semplificare il processo di sviluppo tramite un pacchetto di distribuzione. Usa	• Documentazione ufficiale • Dimensionamento • dispiegamento • tempi di esecuzione • Tutorial sul risolutore e Lambda • Modello di prezzo

Origine dati	Description	Vantaggi	Informazioni supplementari
		un pacchetto per caricare il codice della funzione nel servizio Lambda. Puoi quindi utilizzarlo e i loro ambienti di runtime per testare ed eseguire le tue funzioni. • Versatilità: Lambda può essere utilizzato in una moltitudine di casi d'uso. Puoi integrare senza problemi Lambda con servizi AWS e servizi di terze parti. Alcuni esempi includono pipeline CI/CD e servizi di mailing di massa. • Integrazione con AWS AppSync: puoi richiamare e facilmente le funzioni Lambda nel tuo resolver per gestire le richieste. Il nostro servizio fornisce un'operazione di richiesta semplificata per eseguire chiamate	

Origine dati	Description	Vantaggi	Informazioni supplementari
		Lambda. Consente di chiamare singole e in batch.	

Origine dati	Description	Vantaggi	Informazioni supplementari
OpenSearch	«Amazon OpenSearch Service è un servizio gestito che semplifica l'implementazione, il funzionamento e la scalabilità OpenSearch dei cluster nel AWS cloud. Amazon OpenSearch Service supporta un sistema operativo OpenSearch Elasticsearch legacy (fino alla versione 7.10, l'ultima versione open source del software). Quando si crea un cluster, è possibile scegliere il motore di ricerca da usare. OpenSearch è un motore di ricerca e analisi completamente open source per casi d'uso come l'analisi dei log, il monitoraggio delle applicazioni in tempo reale e l'analisi dei clickstream. Per ulteriori informazioni, consulta la documentazione	• Scalabilità: è possibile scalare facilmente il servizio per adattarlo ai requisiti di servizio tramite OpenSearch Serverless. • Inserimento dei dati: è possibile utilizzare OpenSearch Ingestion per importare, elaborare e analizzare i dati. Esistono molte applicazioni per l'ingestione dei dati, che puoi trovare qui. • Sicurezza: OpenSearch può gestire la configurazione AWS di sicurezza, inclusi IAM CloudTrail, VPCs, autenticazione, ecc. • Disponibilità: supporta OpenSearch anche in diverse regioni e	• Documentazione ufficiale • Serverless • Modello di prezzo

Origine dati	Description	Vantaggi	Informazioni supplementari
	relativa ad OpenSearch h. Amazon OpenSearch Service fornisce tutte le risorse per il OpenSearch cluster e lo avvia. Inoltre, rileva e sostituisce automaticamente i nodi di OpenSearch servizio guasti, riducendo il sovraccarico associato alle infrastrutture autogestite. Puoi scalare il tuo cluster con una singola chiamata API o pochi clic nella console».	zone di disponibilità nel suo servizio. • Integrazione con AWS AppSync: In AWS AppSync, puoi utilizzare GraphQL APIs per archiviare e recuperare dati dai domini di OpenSearch servizio esistenti nel tuo account.	

Origine dati	Description	Vantaggi	Informazioni supplementari
Endpoint HTTP	Puoi utilizzare gli endpoint HTTP come fonti di dati. AWS AppSync può inviare richieste agli endpoint con le informazioni pertinenti come parametri e payload. La risposta HTTP verrà esposta al resolver, che restituirà la risposta finale al termine delle sue operazioni.	• Utile per applicazioni semplici che non sono così integrate con servizi come Lambda.	• Riferimento al resolver

Origine dati	Description	Vantaggi	Informazioni supplementari
Amazon EventBridge	"EventBridge è un servizio serverless che utilizza gli eventi per connettere tra loro i componenti delle applicazioni, semplificando la creazione di applicazioni scalabili basate sugli eventi. Utilizzatelo per indirizzare gli eventi da fonti quali applicazioni, AWS servizi e software di terze parti sviluppati internamente alle applicazioni destinate ai consumatori all'interno dell'organizzazione. EventBridge offre un modo semplice e coerente per importare, filtrare, trasformare e fornire eventi in modo da poter creare nuove applicazioni rapidamente».	• Architettura basata sugli eventi: puoi sfruttare l'architettura basata sugli eventi. • Pianificazione: puoi utilizzare lo EventBridge Scheduler per automatizzare le attività e le regole utilizzando espressioni cron o impostare intervalli di tempo come alternativa ai modelli di eventi. • Pipes: utilizzando EventBridge Pipes, è possibile sostituire il bus degli eventi con una pipe che include modelli di eventi di filtraggio aggiuntivi e l'arricchimento tramite trasformazioni dei dati prima di inviare l'evento alla destinazione. • Integrazione con AWS AppSync: AWS AppSync	• Documentazione ufficiale • Tubi • Pianificatore • Riferimento al resolver • Modello di prezzo

Origine dati	Description	Vantaggi	Informazioni supplementari
		consente di inviare eventi ai bus degli eventi utilizzando il resolver.	

Origine dati	Description	Vantaggi	Informazioni supplementari
Database relazionali	«Amazon Relational Database Service (Amazon RDS) è un servizio Web che semplifica la configurazione, il funzionamento e la scalabilità di un database relazionale nel cloud. AWS Fornisce una capacità ridimensionabile e conveniente per un database relazionale standard del settore e gestisce le attività comuni di amministrazione dei database».	• Gestione semplificata: Periodicamente, RDS esegue la manutenzione delle proprie risorse. La manutenzione prevede più spesso aggiornamenti all'hardware sottostante dell'istanza DB, al sistema operativo (OS) sottostante o alla versione del motore di database. In circostanze normali, puoi decidere quando eseguire gli aggiornamenti (le eccezioni includono le patch di sicurezza). • Consigli: la funzione di raccomandazione di RDS fornisce suggerimenti automatici per risolvere potenziali problemi nell'istanza. • Disponibilità: RDS è disponibile in	• Documentazione ufficiale • Funzionalità • Maintenance (Manutenzione) • Raccomandazioni • Opzioni di archiviazione • Disponibilità • Sicurezza • Modello di prezzo

Origine dati	Description	Vantaggi	Informazioni supplementari
		diverse regioni fisiche in tutto il mondo. Puoi distribuire facilmente le tue esigenze di database su diversi nodi per fornire un servizio migliore ai tuoi clienti. • Personalizzazione: RDS è progettato su misura per soddisfare i requisiti delle grandi aziende. RDS offre diverse opzioni per l'elaborazione, l'implementazione rapida, la scalabilità e l'archiviazione. • Sicurezza: RDS è integrato con diversi strumenti e servizi per mantenere la sicurezza del database a livello di utente, database e rete. • Integrazione con AWS AppSync: se stai cercando una soluzione di backend matura,	

Origine dati	Description	Vantaggi	Informazioni supplementari
		ti AWS AppSync consente di inviare, elaborare, archiviare e restituire dati utilizzando l'istanza come fonte di dati.	
Nessuna fonte di dati	Se non hai intenzione di utilizzare un servizio di origine dati, puoi impostarlo su none. Una fonte di origine dati, sebbene sia ancora esplicitamente classificata come fonte di dati, non è un supporto di archiviazione. Nonostante ciò, è ancora utile in alcuni casi per la manipolazione e il trasferimento dei dati.	- Potenzialmente utile per cose come la conversione dei dati - Utile per risolvere qualcosa a livello locale	Riferimento al resolver

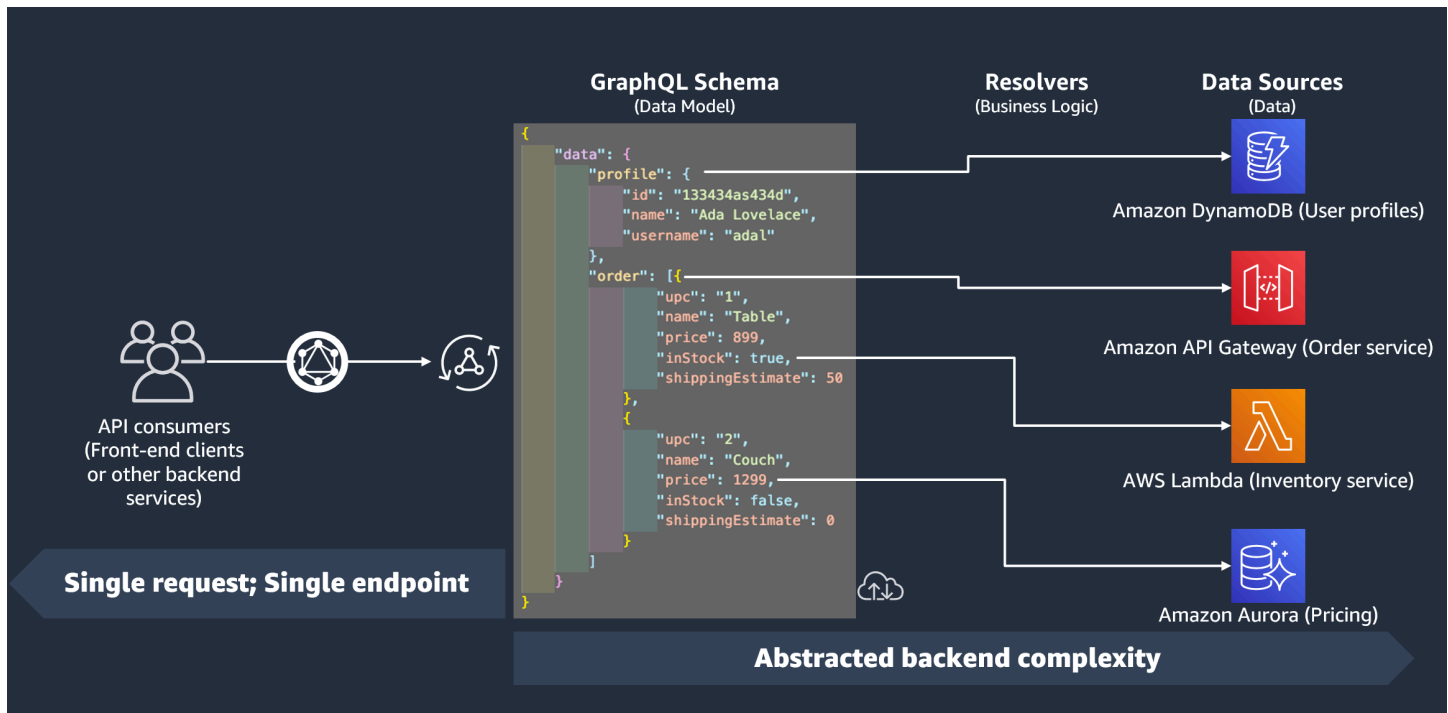
Tip

Per ulteriori informazioni su come interagiscono le sorgenti di dati AWS AppSync, consulta [Allegare un'origine dati](#).

Risolutori

Nelle sezioni precedenti, hai appreso i componenti dello schema e dell'origine dati. Ora, dobbiamo affrontare il modo in cui lo schema e le fonti di dati interagiscono. Tutto inizia con il resolver.

Un resolver è un'unità di codice che gestisce il modo in cui i dati di quel campo verranno risolti quando viene effettuata una richiesta al servizio. I resolver sono associati a campi specifici all'interno dei tipi dello schema. Sono più comunemente usati per implementare le operazioni di modifica dello stato per le operazioni sui campi di interrogazione, mutazione e sottoscrizione. Il resolver elaborerà la richiesta del client, quindi restituirà il risultato, che può essere un gruppo di tipi di output come oggetti o scalari:



Runtime del resolver

In AWS AppSync, devi prima specificare un runtime per il tuo resolver. Un runtime del resolver indica l'ambiente in cui viene eseguito un resolver. Determina anche la lingua in cui verranno scritti i resolver. AWS AppSync attualmente supporta APPSYNC_JS for JavaScript e Velocity Template Language (VTL). [Vedi le funzionalità JavaScript di runtime per i resolver e le funzioni per o il riferimento all'utilità dei modelli di mappatura Resolver per JavaScript VTL.](#)

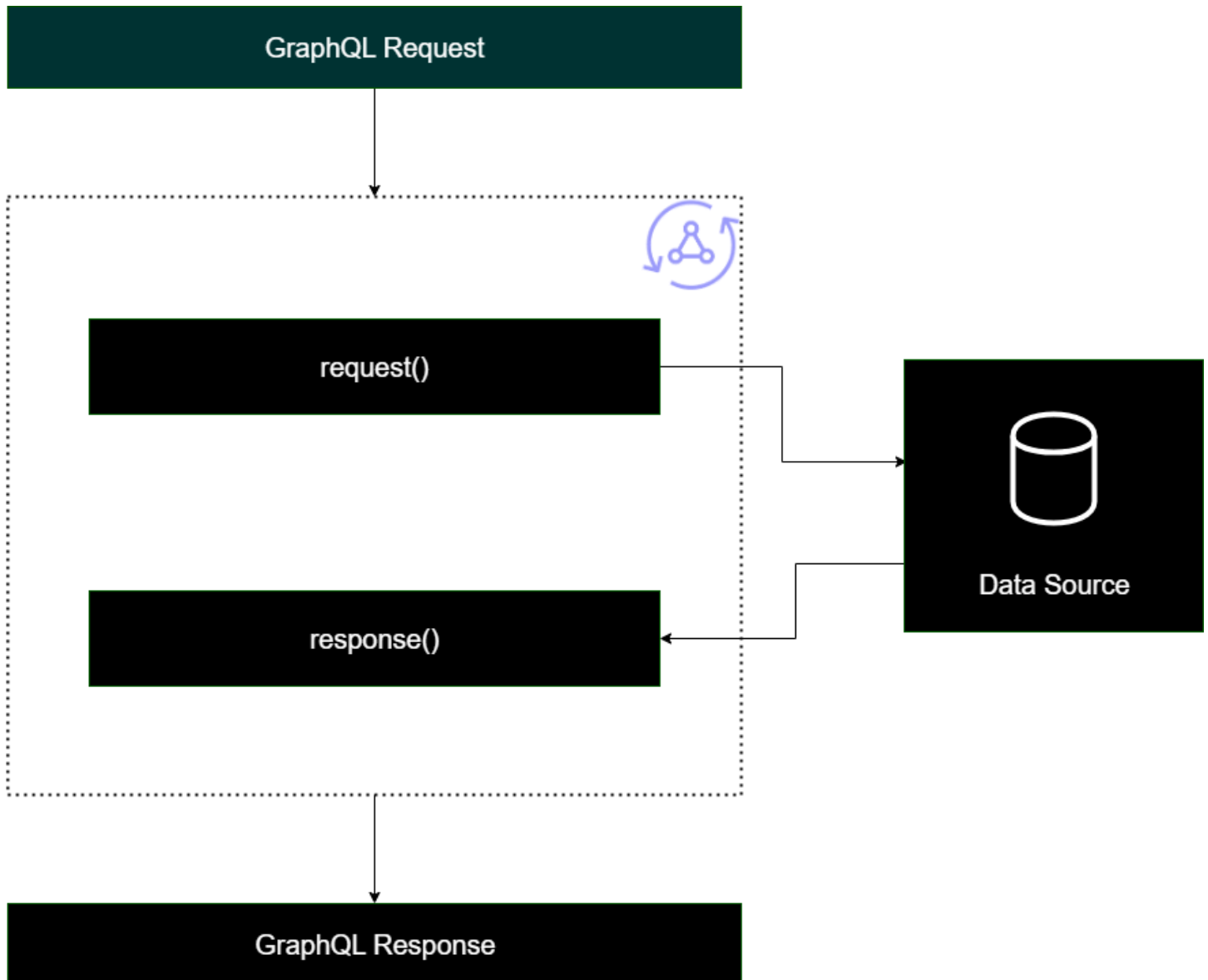
Struttura del resolver

Dal punto di vista del codice, i resolver possono essere strutturati in un paio di modi. Esistono resolver di unità e pipeline.

Resolver di unità

Un resolver di unità è composto da codice che definisce un singolo gestore di richieste e risposte che vengono eseguiti su un'origine dati. Il gestore di richieste accetta un oggetto di contesto come

argomento e restituisce il payload della richiesta utilizzato per chiamare l'origine dei dati. Il gestore della risposta riceve un payload dall'origine dati con il risultato della richiesta eseguita. Il gestore di risposte trasforma il payload in una risposta GraphQL per risolvere il campo GraphQL.

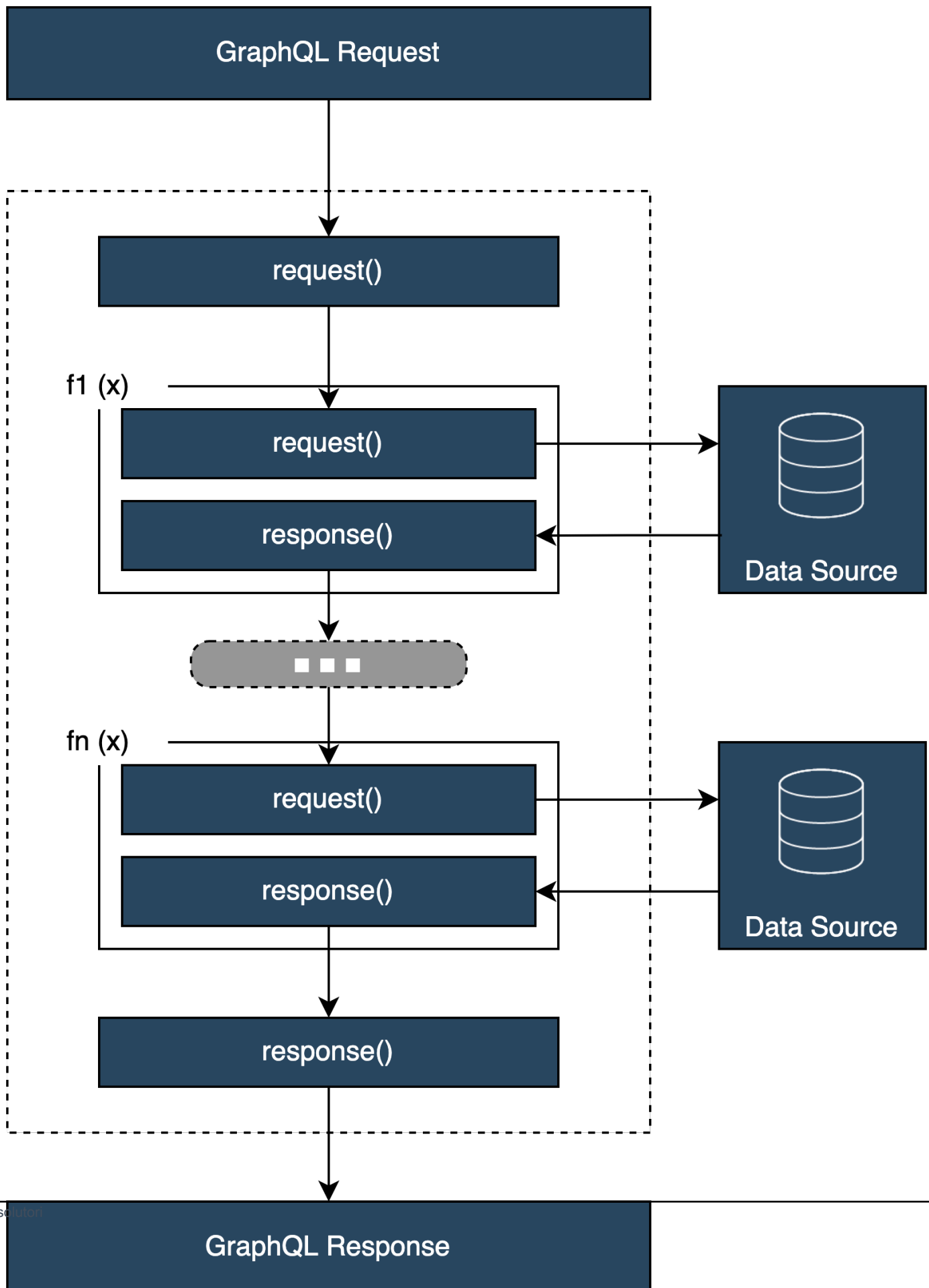


Resolver per pipeline

Quando si implementano i risolutori di pipeline, esiste una struttura generale che seguono:

- Prima della fase: quando viene effettuata una richiesta dal client, ai resolver per i campi dello schema utilizzati (in genere le query, le mutazioni, le sottoscrizioni) vengono trasmessi i dati della richiesta. Il resolver inizierà a elaborare i dati della richiesta con un gestore del passaggio precedente, che consente di eseguire alcune operazioni di preelaborazione prima che i dati passino attraverso il resolver.

- **Funzione/i:** dopo l'esecuzione del passaggio precedente, la richiesta viene passata all'elenco delle funzioni. La prima funzione dell'elenco verrà eseguita sulla fonte di dati. Una funzione è un sottoinsieme del codice del resolver contenente il proprio gestore di richieste e risposte. Un gestore di richieste raccoglierà i dati della richiesta ed eseguirà operazioni sulla fonte dei dati. Il gestore della risposta elaborerà la risposta dell'origine dati prima di restituirla all'elenco. Se è presente più di una funzione, i dati della richiesta verranno inviati alla funzione successiva nell'elenco da eseguire. Le funzioni nell'elenco verranno eseguite in serie nell'ordine definito dallo sviluppatore. Una volta eseguite tutte le funzioni, il risultato finale viene passato alla fase successiva.
- **Dopo la fase:** la fase successiva è una funzione di gestione che consente di eseguire alcune operazioni finali sulla risposta finale della funzione prima di passarla alla risposta GraphQL.



Struttura del gestore Resolver

I gestori sono in genere funzioni chiamate e: Request Response

```
export function request(ctx) {  
  // Code goes here  
}  
  
export function response(ctx) {  
  // Code goes here  
}
```

In un risolutore di unità, ci sarà solo un set di queste funzioni. In un resolver a pipeline, ci sarà un set di queste per la fase prima e dopo e un set aggiuntivo per funzione. Per vedere come potrebbe apparire, esaminiamo un tipo semplice: Query

```
type Query {  
  helloWorld: String!  
}
```

Questa è una semplice interrogazione con un campo chiamato `helloWorld` of type `String`. Supponiamo di voler sempre che questo campo restituisca la stringa «Hello World». Per implementare questo comportamento, dobbiamo aggiungere il resolver a questo campo. In un risolutore di unità, potremmo aggiungere qualcosa del genere:

```
export function request(ctx) {  
  return {}  
}  
  
export function response(ctx) {  
  return "Hello World"  
}
```

`request` Può semplicemente essere lasciato vuoto perché non stiamo richiedendo o elaborando dati. Possiamo anche supporre che la nostra fonte di dati sia `None`, indicando che questo codice non deve eseguire alcuna chiamata. La risposta restituisce semplicemente «Hello World». Per testare questo resolver, dobbiamo fare una richiesta utilizzando il tipo di query:

```
query helloWorldTest {
```



```
helloWorld
}
```

Questa è una query chiamata `helloWorldTest` che restituisce il `helloWorld` campo. Quando viene eseguito, il `helloWorld` field resolver esegue e restituisce anche la risposta:

```
{
  "data": {
    "helloWorld": "Hello World"
  }
}
```

Restituire costanti come questa è la cosa più semplice che si possa fare. In realtà, restituirai input, elenchi e altro. Ecco un esempio più complicato:

```
type Book {
  id: ID!
  title: String
}

type Query {
  getBooks: [Book]
}
```

Qui stiamo restituendo un elenco di `Books`. Supponiamo di utilizzare una tabella `DynamoDB` per archiviare i dati dei libri. I nostri gestori potrebbero avere il seguente aspetto:

```
/**
 * Performs a scan on the dynamodb data source
 */
export function request(ctx) {
  return { operation: 'Scan' };
}

/**
 * return a list of scanned post items
 */
export function response(ctx) {
  return ctx.result.items;
}
```

La nostra richiesta ha utilizzato un'operazione di scansione integrata per cercare tutte le voci della tabella, memorizzato i risultati nel contesto e quindi li ha passati alla risposta. La risposta ha preso gli elementi del risultato e li ha restituiti nella risposta:

```
{
  "data": {
    "getBooks": {
      "items": [
        {
          "id": "abcdefgh-1234-1234-abcdefghijkl",
          "title": "book1"
        },
        {
          "id": "aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee",
          "title": "book2"
        },
        ...
      ]
    }
  }
}
```

Contesto del risolutore

In un resolver, ogni fase della catena di gestori deve essere a conoscenza dello stato dei dati dei passaggi precedenti. Il risultato di un gestore può essere memorizzato e passato a un altro come argomento. GraphQL definisce quattro argomenti di base del resolver:

Argomenti di base del resolver	Description
obj, root, parent e così via.	Il risultato del genitore.
args	Gli argomenti forniti al campo nella query GraphQL.
context	Un valore che viene fornito a ogni resolver e contiene importanti informazioni contestuali

Argomenti di base del resolver	Description
	come l'utente attualmente connesso o l'accesso a un database.
info	Un valore che contiene informazioni specifiche e sul campo relative alla query corrente e i dettagli dello schema.

Nel AWS AppSync, l'argomento [context](#) (ctx) può contenere tutti i dati sopra menzionati. È un oggetto creato su richiesta e contiene dati come credenziali di autorizzazione, dati sui risultati, errori, metadati di richiesta, ecc. Il contesto è un modo semplice per i programmatori di manipolare i dati provenienti da altre parti della richiesta. Riprendi questo frammento:

```
/**
 * Performs a scan on the dynamodb data source
 */
export function request(ctx) {
  return { operation: 'Scan' };
}

/**
 * return a list of scanned post items
 */
export function response(ctx) {
  return ctx.result.items;
}
```

Alla richiesta viene fornito il contesto (ctx) come argomento; questo è lo stato della richiesta. Esegue una scansione di tutti gli elementi di una tabella, quindi memorizza il risultato nel contesto in `result`. Il contesto viene quindi passato all'argomento `response`, che accede a `result` e ne restituisce il contenuto.

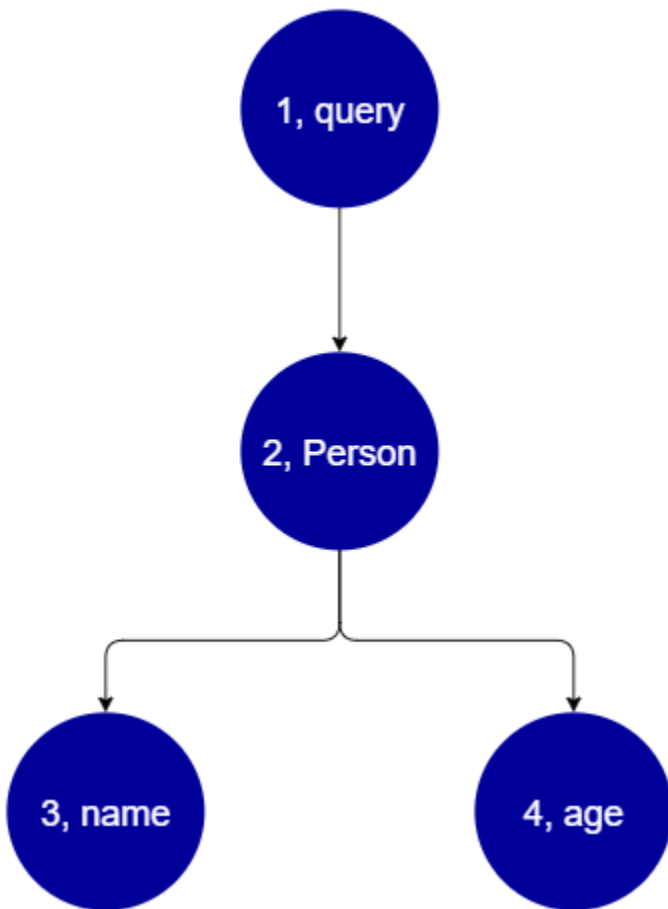
Richieste e analisi

Quando si effettua una query sul servizio GraphQL, questa deve essere sottoposta a un processo di analisi e convalida prima di essere eseguita. La tua richiesta verrà analizzata e tradotta in un albero di sintassi astratto. Il contenuto dell'albero viene convalidato eseguendo diversi algoritmi di convalida rispetto allo schema. Dopo la fase di convalida, i nodi dell'albero vengono attraversati ed elaborati. I

resolver vengono richiamati, i risultati vengono archiviati nel contesto e viene restituita la risposta. Ad esempio, prendiamo questa query:

```
query {  
  Person { //object type  
    name //scalar  
    age  //scalar  
  }  
}
```

Stiamo tornando Person con i campi a e. name age Quando si esegue questa query, l'albero avrà un aspetto simile a questo:



Dall'albero, sembra che questa richiesta cercherà la radice di Query nello schema. All'interno della query, il Person campo verrà risolto. Dagli esempi precedenti, sappiamo che questo potrebbe essere un input dell'utente, un elenco di valori, ecc. Person è molto probabilmente legato a un tipo di oggetto che contiene i campi di cui abbiamo bisogno (nameeage). Una volta trovati, questi due campi secondari vengono risolti nell'ordine indicato (nameseguiti daage). Una volta che l'albero è stato completamente risolto, la richiesta viene completata e verrà rispedita al client.

Proprietà aggiuntive di GraphQL

GraphQL è costituito da diversi principi di progettazione per mantenere la semplicità e la robustezza su larga scala.

Dichiarativo

GraphQL è dichiarativo, il che significa che l'utente descriverà (modellerà) i dati dichiarando solo i campi su cui desidera interrogare. La risposta restituirà solo i dati per queste proprietà. Ad esempio, ecco un'operazione che recupera un Book oggetto in una tabella DynamoDB con il valore ISBN 13 di: **id 9780199536061**

```
{
  getBook(id: "9780199536061") {
    name
    year
    author
  }
}
```

La risposta restituirà i campi nel payload (name, year, e) e author nient'altro:

```
{
  "data": {
    "getBook": {
      "name": "Anna Karenina",
      "year": "1878",
      "author": "Leo Tolstoy",
    }
  }
}
```

Grazie a questo principio di progettazione, GraphQL elimina i problemi perenni di sovra-fetching e insufficiente che REST APIs affronta nei sistemi complessi. Ciò si traduce in una raccolta dati più efficiente e in prestazioni di rete migliorate.

Gerarchico

GraphQL è flessibile in quanto i dati richiesti possono essere modellati dall'utente per soddisfare le esigenze dell'applicazione. I dati richiesti seguono sempre i tipi e la sintassi delle proprietà

definite nell'API GraphQL. Ad esempio, il seguente frammento mostra l'getBookoperazione con un nuovo campo denominato scope quotes che restituisce tutte le stringhe di virgolette e le pagine memorizzate collegate a: Book **9780199536061**

```
{
  getBook(id: "9780199536061") {
    name
    year
    author
    quotes {
      description
      page
    }
  }
}
```

L'esecuzione di questa query restituisce il seguente risultato:

```
{
  "data": {
    "getBook": {
      "name": "Anna Karenina",
      "year": "1878",
      "author": "Leo Tolstoy",
      "quotes": [
        {
          "description": "The highest Petersburg society is essentially one: in it everyone knows everyone else, everyone even visits everyone else.",
          "page": 135
        },
        {
          "description": "Happy families are all alike; every unhappy family is unhappy in its own way.",
          "page": 1
        },
        {
          "description": "To Konstantin, the peasant was simply the chief partner in their common labor.",
          "page": 251
        }
      ]
    }
  }
}
```

```
}
```

Come puoi vedere, i `quotes` campi collegati al libro richiesto sono stati restituiti come matrice nello stesso formato descritto dalla nostra query. Sebbene non sia stato mostrato qui, GraphQL ha l'ulteriore vantaggio di non essere esigente riguardo alla posizione dei dati che sta recuperando. Bookse `quotes` potrebbero essere archiviati separatamente, ma GraphQL recupererà comunque le informazioni finché esiste l'associazione. Ciò significa che la tua query può recuperare moltitudini di dati autonomi in un'unica richiesta.

Introspezione

GraphQL è autodocumentante o introspeettivo. Supporta diverse operazioni integrate che consentono agli utenti di visualizzare i tipi e i campi sottostanti all'interno dello schema. Ad esempio, ecco un `Foo` tipo con un `description` campo `date` and:

```
type Foo {  
  date: String  
  description: String  
}
```

Potremmo usare l'`__type` operazione per trovare i metadati di digitazione sotto lo schema:

```
{  
  __type(name: "Foo") {  
    name                # returns the name of the type  
    fields {            # returns all fields in the type  
      name              # returns the name of each field  
      type {            # returns all types for each field  
        name            # returns the scalar type  
      }  
    }  
  }  
}
```

Ciò restituirà una risposta:

```
{  
  "__type": {  
    "name": "Foo",          # The type name  
    "fields": [  

```

```
{
  "name": "date",          # The date field
  "type": { "name": "String" } # The date's type
},
{
  "name": "description",    # The description field
  "type": { "name": "String" } # The description's type
},
]
}
```

Questa funzionalità può essere utilizzata per scoprire quali tipi e campi sono supportati da un particolare schema GraphQL. GraphQL supporta un'ampia varietà di queste operazioni introspettive.

[Per ulteriori informazioni, consulta Introspection.](#)

Digitazione forte

GraphQL supporta la digitazione avanzata tramite il suo sistema di tipi e campi. Quando definisci qualcosa nel tuo schema, deve avere un tipo che possa essere convalidato prima del runtime. Deve inoltre seguire le specifiche di sintassi di GraphQL. Questo concetto non è diverso dalla programmazione in altri linguaggi. Ad esempio, ecco il Foo tipo precedente:

```
type Foo {
  date: String
  description: String
}
```

Possiamo vedere che Foo è l'oggetto che verrà creato. All'interno di un'istanza di Foo, ci sarà un `date` campo and, entrambi di tipo `String` primitivo (scalare). Sintatticamente, vediamo che Foo è stato dichiarato e i suoi campi esistono all'interno del suo ambito. Questa combinazione di controllo dei tipi e sintassi logica assicura che l'API GraphQL sia concisa e ovvia. [Le specifiche di digitazione e sintassi di GraphQL sono disponibili qui.](#)

Guida introduttiva: creazione della prima API GraphQL in AWS AppSync

È possibile utilizzare la AWS AppSync console per configurare e avviare un'API GraphQL. GraphQL richiede APIs generalmente tre componenti:

1. Schema GraphQL - Lo schema GraphQL è il modello dell'API. Definisce i tipi e i campi che è possibile richiedere quando viene eseguita un'operazione. Per popolare lo schema con dati, devi connettere le fonti di dati all'API GraphQL. In questa guida rapida, creeremo uno schema utilizzando un modello predefinito.
2. Fonti di dati: queste sono le risorse che contengono i dati per popolare l'API GraphQL. Può trattarsi di una tabella DynamoDB, di una funzione Lambda, ecc. Supporta una moltitudine di fonti di dati per creare GraphQL robusto e scalabile AWS AppSync . APIs Le fonti di dati sono collegate ai campi dello schema. Ogni volta che viene eseguita una richiesta su un campo, i dati della fonte popolano il campo. Questo meccanismo è controllato dal resolver. In questa guida rapida, creeremo una fonte di dati utilizzando un modello predefinito insieme allo schema.
3. Risolutori: i resolver sono responsabili del collegamento del campo dello schema all'origine dati. Recuperano i dati dall'origine, quindi restituiscono il risultato in base a ciò che è stato definito dal campo. AWS AppSync supporta entrambi JavaScript e VTL per la scrittura di resolver per GraphQL. APIs In questa guida rapida, i resolver verranno generati automaticamente in base allo schema e alla fonte di dati. Non approfondiremo questo aspetto in questa sezione.

AWS AppSync supporta la creazione e la configurazione di tutti i componenti GraphQL. Quando apri la console, puoi utilizzare i seguenti metodi per creare la tua API:

1. Progettazione di un'API GraphQL personalizzata generandola tramite un modello predefinito e configurando una nuova tabella DynamoDB (origine dati) per supportarla.
2. Progettazione di un'API GraphQL con uno schema vuoto e senza fonti di dati o resolver.
3. Utilizzo di una tabella DynamoDB per importare dati e generare i tipi e i campi dello schema.
4. Utilizzo WebSocket delle funzionalità e AWS AppSync dell' Pub/Sub architettura di Microsoft per lo sviluppo in tempo reale. APIs
5. Utilizzo di GraphQL esistente APIs (source APIs) per il collegamento a un'API unita.

 Note

Consigliamo di rivedere la sezione [Progettazione di uno schema](#) prima di utilizzare strumenti più avanzati. Queste guide spiegheranno esempi più semplici che è possibile utilizzare concettualmente per creare applicazioni più complesse. AWS AppSync

AWS AppSync supporta anche diverse opzioni non da console per creare GraphQL APIs. Ciò include:

1. AWS Amplify
2. AWS SAM
3. CloudFormation
4. Il CDK

L'esempio seguente mostrerà come creare i componenti di base di un'API GraphQL utilizzando modelli predefiniti e DynamoDB.

Argomenti

- [Avvio di uno schema nella console AWS AppSync](#)
- [Facciamo un giro della console AWS AppSync](#)
- [Utilizzo delle mutazioni GraphQL per aggiungere dati a una tabella DynamoDB nella console AWS AppSync](#)
- [Utilizzo delle query GraphQL per recuperare dati da una tabella DynamoDB nella console AWS AppSync](#)
- [Sezioni supplementari per la console AWS AppSync](#)

Avvio di uno schema nella console AWS AppSync

In questo esempio, creerai un'TodoAPI che consente agli utenti di creare Todo articoli per promemoria delle faccende quotidiane come o. *Finish task Pick up groceries* Questa API dimostrerà come utilizzare le operazioni GraphQL in cui lo stato persiste in una tabella DynamoDB.

Concettualmente, ci sono tre passaggi principali per creare la tua prima API GraphQL. È necessario definire lo schema (tipi e campi), collegare le fonti di dati ai campi, quindi scrivere il resolver

che gestisca la logica aziendale. Tuttavia, l'esperienza della console cambia l'ordine. Inizieremo definendo come vogliamo che la nostra fonte di dati interagisca con il nostro schema, quindi definiremo lo schema e il resolver in un secondo momento.

Per creare la tua API GraphQL

1. Accedi a Console di gestione AWS e apri la [AppSync console](#).
2. Nel pannello di controllo, scegliere Create API (Crea API).
3. Mentre GraphQL APIs è selezionato, scegli Design da zero. Quindi, seleziona Successivo.
4. Per il nome dell'API, modifica il nome precompilato in **Todo API**, quindi scegli Avanti.

 Note

Qui sono presenti anche altre opzioni, ma non le useremo per questo esempio.

5. Nella sezione Specificare le risorse GraphQL, procedi come segue:
 - a. Scegli ora Crea tipo supportato da una tabella DynamoDB.

 Note

Ciò significa che creeremo una nuova tabella DynamoDB da allegare come origine dati.

- b. Nel campo Nome modello, immettere. **Todo**

 Note

Il nostro primo requisito è definire il nostro schema. Questo nome del modello sarà il nome del tipo, quindi quello che stai realmente facendo è creare una type chiamata Todo che esisterà nello schema:

```
type Todo {}
```

- c. In Campi, procedi come segue:
 - i. Create un campo denominato **id**, con il tipo ID e il campo obbligatorio impostato su Yes.

 Note

Questi sono i campi che esisteranno nell'ambito del tuo Todo tipo. Il nome del tuo campo qui verrà chiamato `id` con un tipo di `ID!`:

```
type Todo {  
  id: ID!  
}
```

AWS AppSync supporta più valori scalari per diversi casi d'uso.

- ii. Utilizzando **Aggiungi nuovo campo**, crea quattro campi aggiuntivi con i Name valori impostati su **name**, **when**, **where**, **edescription**. Typel loro valori saranno `String`, e i Required valori `Array` and saranno entrambi impostati su `No`. Corrisponderà a quanto segue:

Model information

Model name

A model is a type with preconfigured queries, mutations, and subscriptions.

Todo

The model name must have 1 to 50 characters. Valid characters: A-Z, a-z, 0-9, and _

Fields

Models have fields. Fields have a name and a type.

Name	Type	Array	Required	
id	ID ▼	No ▼	Yes ▼	Remove
name	String ▼	No ▼	No ▼	Remove
when	String ▼	No ▼	No ▼	Remove
where	String ▼	No ▼	No ▼	Remove
description	String ▼	No ▼	No ▼	Remove
Add new field				

 Note

Il tipo completo e i relativi campi avranno il seguente aspetto:

```
type Todo {  
  id: ID!  
  name: String  
  when: String  
  where: String  
  description: String  
}
```

Poiché stiamo creando uno schema utilizzando questo modello predefinito, verrà inoltre popolato con diverse mutazioni standard basate sul tipo, ad esempio `create`, `delete`, e `update` per aiutarti a popolare facilmente la tua fonte di dati.

- d. In configura la tabella del modello, inserisci un nome di tabella, ad esempio. **TodoAPITable**. Imposta la chiave primaria su `id`.

 Note

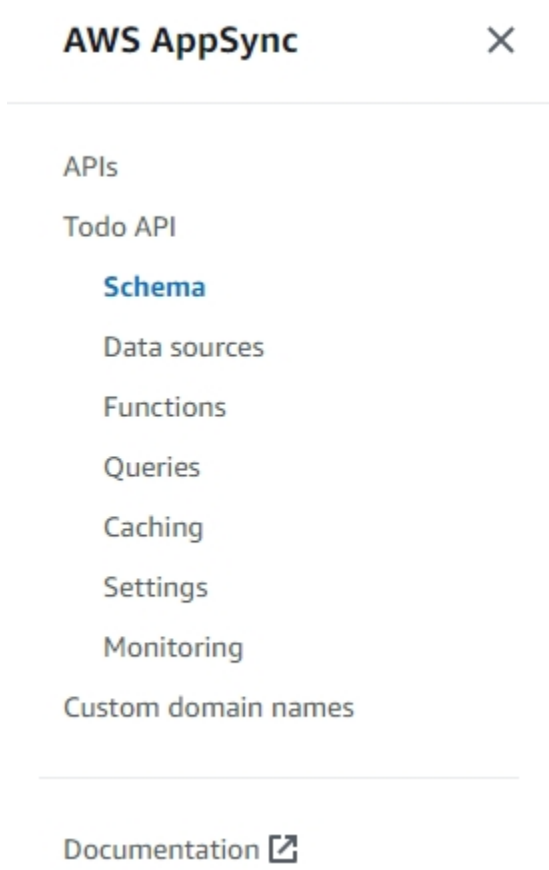
Stiamo essenzialmente creando una nuova tabella DynamoDB *TodoAPITable* chiamata che verrà allegata all'API come fonte di dati principale. La nostra chiave primaria è impostata sul `id` campo richiesto che abbiamo definito in precedenza. Nota che questa nuova tabella è vuota e non contiene nulla tranne la chiave di partizione.

- e. Scegli Next (Successivo).
6. Controlla le modifiche e scegli Crea API. Attendi un momento per consentire al AWS AppSync servizio di completare la creazione della tua API.

Hai creato con successo un'API GraphQL con il relativo schema e l'origine dati DynamoDB. Per riassumere i passaggi precedenti, abbiamo scelto di creare un'API GraphQL completamente nuova. Abbiamo definito il nome dell'API, quindi aggiunto la definizione dello schema aggiungendo il nostro primo tipo. Abbiamo definito il tipo e i relativi campi, quindi abbiamo scelto di allegare un'origine dati a uno dei campi creando una nuova tabella DynamoDB senza dati.

Facciamo un giro della console AWS AppSync

Prima di aggiungere dati alla nostra tabella DynamoDB, dobbiamo esaminare le funzionalità di base dell'esperienza della AWS AppSync console. La scheda della AWS AppSync console sul lato sinistro della pagina consente agli utenti di navigare facilmente tra i principali componenti o opzioni di configurazione che forniscono: AWS AppSync



Progettista dello schema

Scegli Schema per visualizzare lo schema appena creato. Se esami il contenuto dello schema, noterai che è già stato caricato con una serie di operazioni di supporto per semplificare il processo di sviluppo. Nell'editor dello schema, se scorrete il codice, alla fine raggiungerete il modello definito nella sezione precedente:

```
type Todo {  
  id: ID!  
  name: String  
  when: String  
  where: String
```

```
description: String
}
```

Il modello è diventato il tipo di base utilizzato in tutto lo schema. Inizieremo ad aggiungere dati alla nostra fonte di dati utilizzando mutazioni generate automaticamente da questo tipo.

Ecco alcuni suggerimenti e informazioni aggiuntivi sull'editor dello schema:

1. L'editor di codice dispone di funzionalità di linting e controllo degli errori che puoi usare quando scrivi le tue app.
2. Sul lato destro della console vengono mostrati i tipi GraphQL che sono stati creati, nonché i resolver nei diversi tipi di primo livello, ad esempio le query.
3. Quando aggiungi nuovi tipi a uno schema (ad esempio, `type User { ... }`), puoi fare in modo che le risorse AWS AppSync DynamoDB vengano fornite automaticamente. Tali risorse includono la chiave primaria, la chiave di ordinamento e la progettazione dell'indice appropriate che corrispondono al meglio al modello di accesso ai dati GraphQL. Se scegli Create Resources (Crea risorse) nella parte superiore e selezioni uno di questi tipi definiti dall'utente nel menu a discesa, puoi scegliere diverse opzioni di campo nella progettazione di uno schema. Tratteremo questo aspetto nella sezione relativa alla [progettazione di uno schema](#).

Configurazione del resolver

Nella finestra di progettazione dello schema, la sezione Resolver contiene tutti i tipi e i campi dello schema. Se scorri l'elenco dei campi, noterai che puoi allegare i resolver a determinati campi scegliendo Allega. Si aprirà un editor di codice in cui potrai scrivere il codice del resolver. AWS AppSync supporta sia VTL che i JavaScript runtime, che possono essere modificati nella parte superiore della pagina scegliendo Azioni, quindi Update Runtime. Nella parte inferiore della pagina, puoi anche creare funzioni che eseguiranno diverse operazioni in sequenza. Tuttavia, i resolver sono un argomento avanzato e non lo tratteremo in questa sezione.

Origini dati

Scegli Origini dati per visualizzare la tabella DynamoDB. Scegliendo l'Resource opzione (se disponibile), puoi visualizzare la configurazione della tua fonte di dati. Nel nostro esempio, questo porta alla console DynamoDB. Da lì, puoi modificare i tuoi dati. Puoi anche modificare direttamente alcuni dati scegliendo la fonte dei dati, quindi scegliendo Modifica. Se hai bisogno di eliminare la tua fonte di dati, puoi scegliere la tua fonte di dati, quindi selezionare Elimina. Infine, puoi creare

nuove fonti di dati scegliendo Crea origine dati, quindi configurando il nome e il tipo. Nota che questa opzione serve per collegare il AWS AppSync servizio a una risorsa esistente. Devi comunque creare la risorsa nel tuo account utilizzando il servizio pertinente prima di AWS AppSync riconoscerla.

Query

Scegli Query per visualizzare le tue domande e le tue mutazioni. Quando abbiamo creato la nostra API GraphQL utilizzando il nostro modello, abbiamo generato AWS AppSync automaticamente alcune mutazioni e query di supporto a scopo di test. Nell'editor di query, il lato sinistro contiene Explorer. Questo è un elenco che mostra tutte le tue mutazioni e le tue domande. Puoi abilitare facilmente le operazioni e i campi che desideri utilizzare qui facendo clic sui valori dei relativi nomi. Ciò farà sì che il codice appaia automaticamente nella parte centrale dell'editor. Qui puoi modificare le tue mutazioni e le tue query modificando i valori. Nella parte inferiore dell'editor, hai l'editor Query Variable che ti permette di inserire i valori dei campi per le variabili di input delle tue operazioni. Scegliendo Esegui nella parte superiore dell'editor verrà visualizzato un elenco a discesa per selezionare il file da query/mutation eseguire. L'output di questa esecuzione verrà visualizzato sul lato destro della pagina. Tornando alla sezione Explorer in alto, puoi scegliere un'operazione (Query, Mutation, Subscription), quindi scegliere il simbolo + per aggiungere una nuova istanza di quella particolare operazione. Nella parte superiore della pagina, ci sarà un altro elenco a discesa che contiene la modalità di autorizzazione per le esecuzioni delle query. Tuttavia, non tratteremo questa funzionalità in questa sezione (per ulteriori informazioni, consulta [Sicurezza](#)).

Settings

Scegli Impostazioni per visualizzare alcune opzioni di configurazione per l'API GraphQL. Qui puoi abilitare alcune opzioni come la registrazione, il tracciamento e la funzionalità firewall delle applicazioni web. Puoi anche aggiungere nuove modalità di autorizzazione per proteggere i tuoi dati da fughe indesiderate verso il pubblico. Tuttavia, queste opzioni sono più avanzate e non verranno trattate in questa sezione.

Note

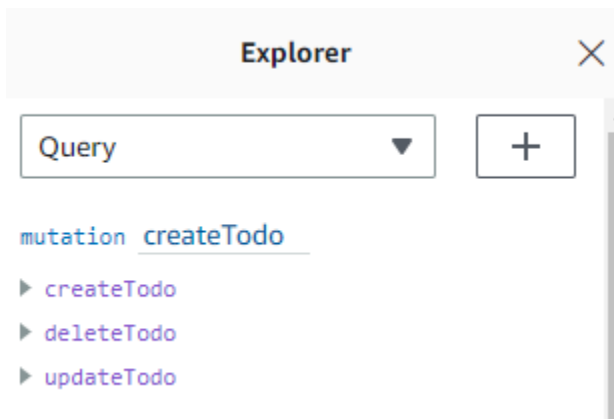
La modalità di autorizzazione predefinita utilizza una chiave API per testare l'applicazione. **API_KEY** Questa è l'autorizzazione di base fornita a tutti i APIs GraphQL appena creati. Ti consigliamo di utilizzare un metodo di produzione diverso. A titolo di esempio in questa sezione, utilizzeremo solo la chiave API. Per ulteriori informazioni sui metodi di autorizzazione supportati, consulta [Sicurezza](#).

Utilizzo delle mutazioni GraphQL per aggiungere dati a una tabella DynamoDB nella console AWS AppSync

Il passaggio successivo consiste nell'aggiungere dati alla tabella DynamoDB vuota utilizzando una mutazione GraphQL. Le mutazioni sono uno dei tipi di operazioni fondamentali in GraphQL. Sono definite nello schema e consentono di manipolare i dati nella fonte dei dati. In termini di REST APIs, sono molto simili a operazioni come PUT o POST.

Per aggiungere dati alla tua fonte di dati

1. Se non l'hai già fatto, accedi Console di gestione AWS e apri la [AppSync console](#).
2. Scegli la tua API dalla tabella.
3. Nella scheda a sinistra, scegli Query.
4. Nella scheda Explorer a sinistra della tabella, potresti vedere diverse mutazioni e query già definite nell'editor di query:



Note

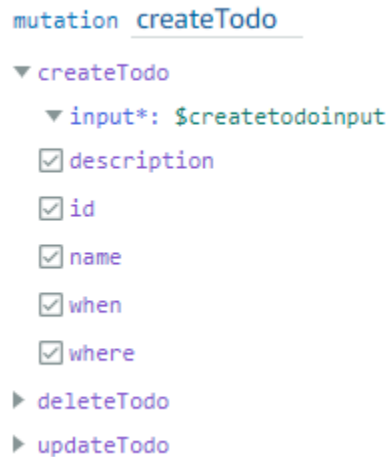
Questa mutazione è effettivamente presente nel tuo schema come tipo. Mutation Ha il codice:

```
type Mutation {  
  createTodo(input: CreateTodoInput!): Todo  
  updateTodo(input: UpdateTodoInput!): Todo  
  deleteTodo(input: DeleteTodoInput!): Todo  
}
```

Come puoi vedere, le operazioni qui sono simili a quelle presenti nell'editor di query.

AWS AppSync li ha generati automaticamente dal modello che abbiamo definito in precedenza. Questo esempio utilizzerà la `createTodo` mutazione per aggiungere voci alla nostra *TodoAPITable* tabella.

5. Scegli l'`createTodo` operazione espandendola sotto la `createTodo` mutazione:



```
mutation createTodo
  ▼ createTodo
    ▼ input*: $createtodoinput
      ☒ description
      ☒ id
      ☒ name
      ☒ when
      ☒ where
    ▶ deleteTodo
    ▶ updateTodo
```

Abilita le caselle di controllo per tutti i campi, come nell'immagine sopra.

Note

Gli attributi che vedete qui sono i diversi elementi modificabili della mutazione. Il tuo input può essere considerato il parametro di `createTodo`. Le varie opzioni con caselle di controllo sono i campi che verranno restituiti nella risposta una volta eseguita un'operazione.

6. Nell'editor di codice al centro dello schermo, noterai che l'operazione appare sotto la mutazione: `createTodo`

```
mutation createTodo($createtodoinput: CreateTodoInput!) {
  createTodo(input: $createtodoinput) {
    where
    when
    name
    id
  }
}
```

```

    description
  }
}
```

Note

Per spiegare correttamente questo frammento, dobbiamo anche guardare il codice dello schema. La dichiarazione `mutation createTodo($createtodoinput: CreateTodoInput!){}` è la mutazione con una delle sue operazioni, `createTodo`. La mutazione completa si trova nello schema:

```

type Mutation {
  createTodo(input: CreateTodoInput!): Todo
  updateTodo(input: UpdateTodoInput!): Todo
  deleteTodo(input: DeleteTodoInput!): Todo
}
```

Tornando alla dichiarazione di mutazione dell'editor, il parametro è un oggetto chiamato `$createtodoinput` con un tipo di input richiesto di `CreateTodoInput`. Nota che `CreateTodoInput` (e tutti gli input della mutazione) sono definiti anche nello schema. Ad esempio, ecco il codice boilerplate per: `CreateTodoInput`

```

input CreateTodoInput {
  name: String!
  when: String!
  where: String!
  description: String!
}
```

Contiene i campi che abbiamo definito nel nostro modello, vale a dire, `name`, `when`, `where` e `description`.

Tornando al codice dell'editor, `createTodo(input: $createtodoinput){}`, dichiariamo l'input come `$createtodoinput`, che è stato utilizzato anche nella dichiarazione di mutazione. Lo facciamo perché ciò consente a GraphQL di convalidare i nostri input rispetto ai tipi forniti e di garantire che vengano utilizzati con gli input corretti. La parte finale del codice dell'editor mostra i campi che verranno restituiti nella risposta dopo l'esecuzione di un'operazione:

```
{
```

```
    where
    when
    name
    id
    description
  }
```

Nella scheda Variabili di interrogazione sotto questo editor, ci sarà un `createtodoinput` oggetto generico che può contenere i seguenti dati:

```
{
  "createtodoinput": {
    "name": "Hello, world!",
    "when": "Hello, world!",
    "where": "Hello, world!",
    "description": "Hello, world!"
  }
}
```

Note

Qui è dove assegniamo i valori per l'input menzionato in precedenza:

```
input CreateTodoInput {
  name: String
  when: String
  where: String
  description: String
}
```

Modifica il `createtodoinput` aggiungendo le informazioni che vogliamo inserire nella nostra tabella DynamoDB. In questo caso, volevamo creare alcuni Todo elementi come promemoria:

```
{
  "createtodoinput": {
    "name": "Shopping List",
    "when": "Friday",
```

```
"where": "Home",
"description": "I need to buy eggs"
}
}
```

7. Scegli Esegui nella parte superiore dell'editor. Scegli CreateToDo nell'elenco a discesa. Sul lato destro dell'editor, dovresti vedere la risposta. Potrebbe essere simile a quanto segue:

```
{
  "data": {
    "createToDo": {
      "where": "Home",
      "when": "Friday",
      "name": "Shopping List",
      "id": "abcdefgh-1234-1234-1234-abcdefghijkl",
      "description": "I need to buy eggs"
    }
  }
}
```

Se accedi al servizio DynamoDB, ora vedrai una voce nella tua origine dati con queste informazioni:

TodoAPITable

▶

Scan or query items

Expand to query or scan items.

✔

Completed. Read capacity units consumed: 2

Items returned (1)

☐	id	description	name	when	where
☐		I need to buy ...	Shopping List	Friday	Home

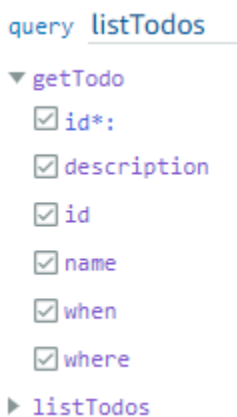
Per riassumere l'operazione, il motore GraphQL ha analizzato il record e un resolver lo ha inserito nella tabella Amazon DynamoDB. Ancora una volta, puoi verificarlo nella console DynamoDB. Nota che non è necessario inserire un `id` valore. Un `id` viene generato e restituito nei risultati. Questo perché l'esempio ha utilizzato una `autoId()` funzione in un resolver GraphQL per la chiave di partizione impostata sulle risorse DynamoDB. Tratteremo come creare resolver in una sezione diversa. Prendi nota del `id` valore restituito; lo utilizzerai nella sezione successiva per recuperare i dati con una query GraphQL.

Utilizzo delle query GraphQL per recuperare dati da una tabella DynamoDB nella console AWS AppSync

Ora che esiste un record nel database, otterrete risultati quando eseguite una query. Una query è una delle altre operazioni fondamentali di GraphQL. Viene utilizzato per analizzare e recuperare informazioni dalla fonte di dati. In termini di REST APIs, questa è simile all'GET operazione. Il vantaggio principale delle query GraphQL è la possibilità di specificare i requisiti esatti dei dati dell'applicazione in modo da recuperare i dati pertinenti al momento giusto.

Per interrogare la fonte dei dati

1. Se non l'hai già fatto, accedi Console di gestione AWS e apri la [AppSync console](#).
2. Scegli la tua API dalla tabella.
3. Nella scheda a sinistra, scegli Query.
4. Nella scheda Explorer a sinistra della tabella, sotto `querylistTodos`, espandi l'`getTodo` operazione:



5. Nell'editor di codice, dovresti vedere il codice dell'operazione:

```
query listTodos {
```

```
getTodo(id: "") {  
  description  
  id  
  name  
  when  
  where  
}
```

Inserisci (`id: ""`) il valore che hai salvato nel risultato dell'operazione di mutazione. Nel nostro esempio, questo sarebbe:

```
query listTodos {  
  getTodo(id: "abcdefgh-1234-1234-1234-abcdefghijkl") {  
    description  
    id  
    name  
    when  
    where  
  }  
}
```

6. Scegli Esegui, quindi ListTodos. Il risultato verrà visualizzato a destra dell'editor. Il nostro esempio si presentava così:

```
{  
  "data": {  
    "getTodo": {  
      "description": "I need to buy eggs",  
      "id": "abcdefgh-1234-1234-1234-abcdefghijkl",  
      "name": "Shopping List",  
      "when": "Friday",  
      "where": "Home"  
    }  
  }  
}
```

Note

Le query restituiscono solo i campi specificati. Puoi deselezionare i campi che non ti servono eliminandoli dal campo di ritorno:

```
{
```

```

    description
    id
    name
    when
    where
  }

```

Puoi anche deselezionare la casella nella scheda Explorer accanto al campo che desideri eliminare.

7. Puoi anche provare l'`listTodos` operazione ripetendo i passaggi per creare una voce nell'origine dati, quindi ripetendo i passaggi di interrogazione con l'operazione. `listTodos` Ecco un esempio in cui abbiamo aggiunto una seconda attività:

```

{
  "createtodoinput": {
    "name": "Second Task",
    "when": "Monday",
    "where": "Home",
    "description": "I need to mow the lawn"
  }
}

```

Richiamando l'`listTodos` operazione, ha restituito sia le voci vecchie che quelle nuove:

```

{
  "data": {
    "listTodos": {
      "items": [
        {
          "id": "abcdefgh-1234-1234-1234-abcdefghijkl",
          "name": "Shopping List",
          "when": "Friday",
          "where": "Home",
          "description": "I need to buy eggs"
        },
        {
          "id": "aaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee",
          "name": "Second Task",
          "when": "Monday",
          "where": "Home",
          "description": "I need to mow the lawn"
        }
      ]
    }
  }
}

```



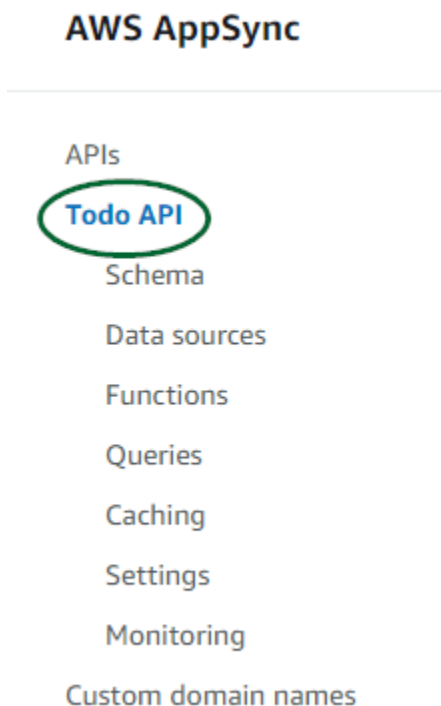
```
}  
  ]  
    }  
  }  
}
```

Sezioni supplementari per la console AWS AppSync

Queste sezioni sono un riferimento per AWS AppSync argomenti più avanzati. Ti consigliamo di seguire la sezione [Lettura supplementare](#) prima di fare qualsiasi altra cosa.

Integrazione

Nella scheda della console, se scegli il nome della tua API, viene visualizzata la pagina Integrazione:



Riassume i passaggi per configurare l'API e delinea i passaggi successivi per la creazione di un'applicazione client. La sezione [Integra con la tua app](#) fornisce dettagli sull'utilizzo della AWS toolchain [Amplify](#) per automatizzare il processo di connessione dell'API con iOS, Android JavaScript e applicazioni tramite la configurazione e la generazione di codice. La toolchain Amplify fornisce supporto completo per la creazione di progetti dalla workstation locale, incluso il provisioning GraphQL e i flussi di lavoro per CI/CD.

La sezione Client Samples elenca anche esempi di applicazioni client (ad esempio JavaScript, iOS, Android) per testare un' end-to-end esperienza. Puoi clonare e scaricare questi esempi e il file di configurazione contiene le informazioni necessarie (come l'URL dell'endpoint) necessarie per iniziare. Segui le istruzioni sulla pagina della [AWS Amplify toolchain](#) per eseguire l'app.

Lettura supplementare

- [Progettazione GraphQL APIs](#)- Questa è una guida completa per creare GraphQL utilizzando uno schema vuoto senza fonti di dati o resolver.

Progettazione di GraphQL APIs con AWS AppSync

AWS AppSync consente di creare GraphQL APIs utilizzando l'esperienza della console. L'hai visto di sfuggita nella sezione [Avvio](#) di uno schema di esempio. Tuttavia, quella guida non mostrava l'intero catalogo di opzioni e configurazioni che potevi sfruttare. AWS AppSync

Quando scegli di creare un'API GraphQL nella console, ci sono diverse opzioni da esplorare. Se hai seguito la nostra guida all'[avvio di uno schema di esempio](#), ti abbiamo mostrato come creare un'API da un modello predefinito. Nelle sezioni seguenti, ti guideremo attraverso il resto delle opzioni e configurazioni per la creazione di APIs AWS AppSync GraphQL in.

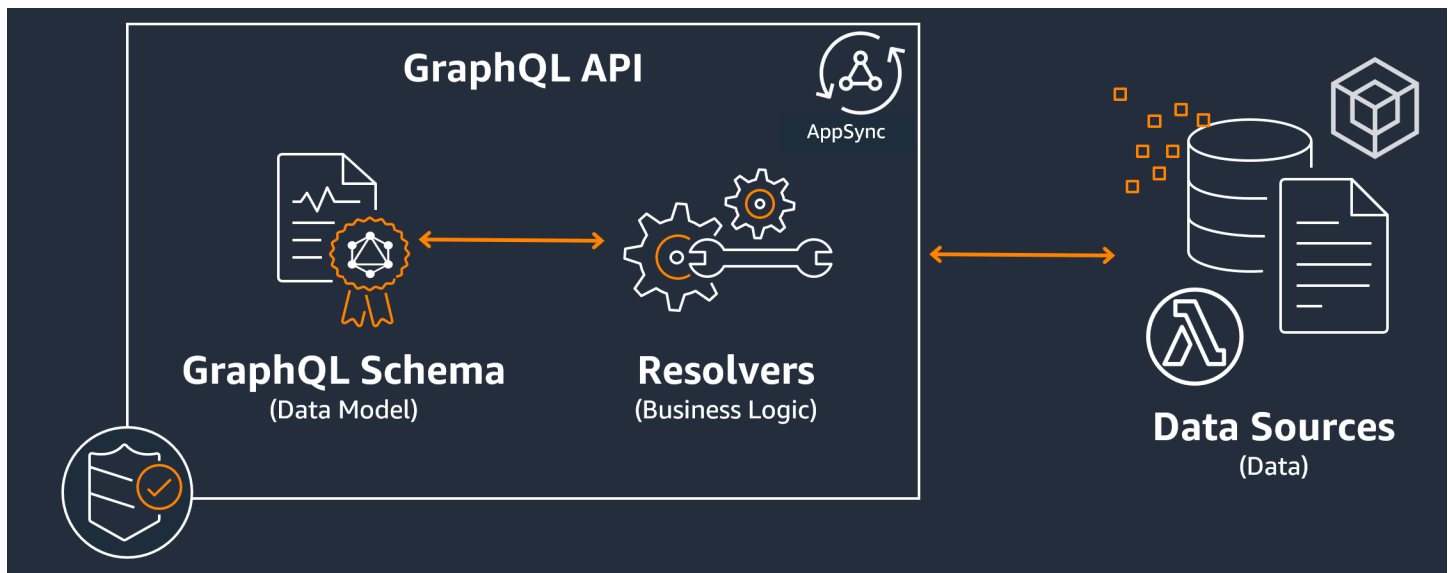
In questa sezione, esaminerai i seguenti concetti:

1. [Blank APIs or imports](#): Questa guida illustrerà l'intero processo di creazione di un'API GraphQL. Imparerai come creare un GraphQL da un modello vuoto senza modello, configurare le fonti di dati per il tuo schema e aggiungere il tuo primo resolver a un campo.
2. [Real-time data](#): Questa guida ti mostrerà le potenziali opzioni per creare un'API utilizzando AWS AppSync il motore di G's. WebSocket
3. [Merged APIs](#): Questa guida ti mostrerà come creare un nuovo GraphQL associando e unendo dati APIs da più GraphQL esistenti. APIs
4. [the section called “Creazione di GraphQL APIs con l'introspezione RDS”](#): Questa guida ti mostrerà come integrare le tue tabelle Amazon RDS utilizzando un'API di dati.

Strutturazione di un'API GraphQL (vuota o importata) APIs

Prima di creare l'API GraphQL da un modello vuoto, sarebbe utile esaminare i concetti relativi a GraphQL. Esistono tre componenti fondamentali di un'API GraphQL:

1. Lo schema è il file contenente la forma e la definizione dei dati. Quando un client effettua una richiesta al servizio GraphQL, i dati restituiti seguiranno le specifiche dello schema. Per ulteriori informazioni, consulta [Schemi GraphQL](#).
2. L'origine dati è allegata allo schema. Quando viene effettuata una richiesta, è qui che i dati vengono recuperati e modificati. Per ulteriori informazioni, consulta [Data sources](#).
3. Il resolver si trova tra lo schema e l'origine dati. Quando viene effettuata una richiesta, il resolver esegue l'operazione sui dati dall'origine, quindi restituisce il risultato come risposta. Per ulteriori informazioni, consulta [Resolvers](#).



AWS AppSync gestisce il tuo APIs consentendoti di creare, modificare e archiviare il codice per i tuoi schemi e resolver. Le tue fonti di dati proverranno da repository esterni come database, tabelle DynamoDB e funzioni Lambda. Se utilizzi un AWS servizio per archiviare i tuoi dati o hai intenzione di farlo, AWS AppSync offre un'esperienza quasi perfetta quando associ i dati dei tuoi account AWS al tuo GraphQL. APIs

Nella sezione successiva, imparerai come creare ciascuno di questi componenti utilizzando il servizio. AWS AppSync

Argomenti

- [Progettazione dello schema GraphQL](#)
- [Collegamento di un'origine dati in AWS AppSync](#)
- [Configurazione dei resolver in AWS AppSync](#)
- [Utilizzo di un' AWS AppSync API con AWS CDK](#)

Progettazione dello schema GraphQL

Lo schema GraphQL è alla base di qualsiasi implementazione del server GraphQL. Ogni API GraphQL è definita da un unico schema che contiene tipi e campi che descrivono come verranno popolati i dati delle richieste. I dati che fluiscono attraverso l'API e le operazioni eseguite devono essere convalidati rispetto allo schema.

In generale, il [sistema di tipo GraphQL](#) descrive le funzionalità di un server GraphQL e viene utilizzato per determinare se una query è valida. Il sistema di tipi di server viene spesso definito schema del

server e può essere costituito da diversi tipi di oggetti, tipi scalari, tipi di input e altro ancora. GraphQL è sia dichiarativo che fortemente tipizzato, il che significa che i tipi saranno ben definiti in fase di esecuzione e restituiranno solo ciò che è stato specificato.

AWS AppSync consente di definire e configurare schemi GraphQL. La sezione seguente descrive come creare schemi GraphQL da zero utilizzando AWS AppSync i servizi di GraphQL.

Strutturazione di uno schema GraphQL

Tip

Consigliamo di rivedere la sezione [Schemas](#) prima di continuare.

GraphQL è un potente strumento per l'implementazione di servizi API. Secondo il [sito Web di GraphQL](#), [GraphQL è](#) il seguente:

«GraphQL è un linguaggio di query APIs e un runtime per soddisfare tali richieste con i dati esistenti. GraphQL fornisce una descrizione completa e comprensibile dei dati dell'API, offre ai clienti la possibilità di chiedere esattamente ciò di cui hanno bisogno e nient'altro, facilita l'evoluzione APIs nel tempo e abilita potenti strumenti di sviluppo. »

Questa sezione tratta la primissima parte dell'implementazione GraphQL, lo schema. Utilizzando la citazione precedente, uno schema svolge il ruolo di «fornire una descrizione completa e comprensibile dei dati nell'API». In altre parole, uno schema GraphQL è una rappresentazione testuale dei dati, delle operazioni e delle relazioni tra di essi del servizio. Lo schema è considerato il punto di ingresso principale per l'implementazione del servizio GraphQL. Non sorprende che sia spesso una delle prime cose che fai nel tuo progetto. Ti consigliamo di rivedere la sezione [Schemi prima di continuare](#).

Per citare la sezione [Schemas](#), gli schemi GraphQL sono scritti nello Schema Definition Language (SDL). SDL è composto da tipi e campi con una struttura consolidata:

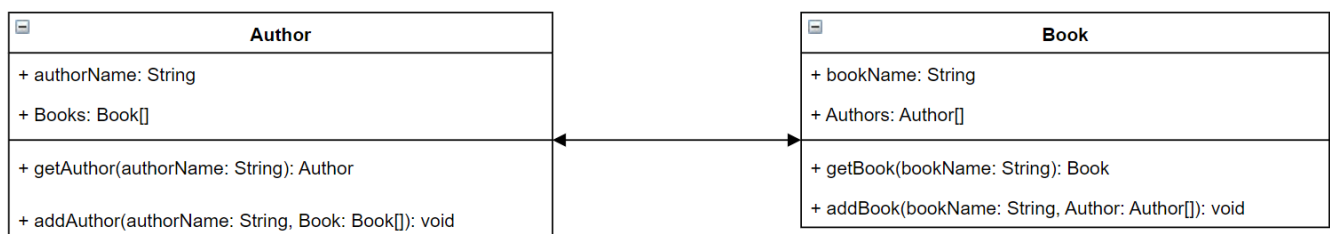
- **Tipi:** I tipi sono il modo in cui GraphQL definisce la forma e il comportamento dei dati. GraphQL supporta una moltitudine di tipi che verranno spiegati più avanti in questa sezione. Ogni tipo definito nello schema conterrà il proprio ambito. All'interno dell'ambito ci saranno uno o più campi che possono contenere un valore o una logica che verrà utilizzata nel servizio GraphQL. I tipi ricoprono molti ruoli diversi, i più comuni sono gli oggetti o gli scalari (tipi di valori primitivi).

- **Campi:** i campi esistono nell'ambito di un tipo e contengono il valore richiesto dal servizio GraphQL. Sono molto simili alle variabili di altri linguaggi di programmazione. La forma dei dati definiti nei campi determinerà il modo in cui i dati sono strutturati in un' request/response operazione. Ciò consente agli sviluppatori di prevedere cosa verrà restituito senza sapere come viene implementato il backend del servizio.

Gli schemi più semplici conterranno tre diverse categorie di dati:

1. **Radici dello schema:** le radici definiscono i punti di ingresso dello schema. Indica i campi che eseguiranno alcune operazioni sui dati come aggiungere, eliminare o modificare qualcosa.
2. **Tipi:** si tratta di tipi di base utilizzati per rappresentare la forma dei dati. Puoi quasi pensarli come oggetti o rappresentazioni astratte di qualcosa con caratteristiche definite. Ad esempio, è possibile creare un `Person` oggetto che rappresenti una persona in un database. Le caratteristiche di ogni persona verranno definite all'interno dei campi `Person` as. Possono essere qualsiasi cosa, ad esempio il nome, l'età, il lavoro, l'indirizzo della persona, ecc.
3. **Tipi di oggetti speciali:** questi sono i tipi che definiscono il comportamento delle operazioni nello schema. Ogni tipo di oggetto speciale viene definito una volta per schema. Vengono prima inseriti nella radice dello schema, quindi definiti nel corpo dello schema. Ogni campo di un tipo di oggetto speciale definisce una singola operazione che deve essere implementata dal resolver.

Per metterlo in prospettiva, immagina di creare un servizio che memorizza gli autori e i libri che hanno scritto. Ogni autore ha un nome e una serie di libri di cui è autore. Ogni libro ha un nome e un elenco di autori associati. Vogliamo anche avere la possibilità di aggiungere o recuperare libri e autori. Una semplice rappresentazione UML di questa relazione può avere il seguente aspetto:



In GraphQL, le entità `Author` e `Book` rappresentano due diversi tipi di oggetti nello schema:

```
type Author {
```

```
}

type Book {
}
```

Author contiene `authorName` e `Books`, mentre `Book` contiene `bookName` e `Authors`. Questi possono essere rappresentati come campi che rientrano nell'ambito dei tuoi tipi:

```
type Author {
  authorName: String
  Books: [Book]
}

type Book {
  bookName: String
  Authors: [Author]
}
```

Come potete vedere, le rappresentazioni dei tipi sono molto simili al diagramma. Tuttavia, i metodi sono quelli in cui la cosa diventa un po' più complicata. Questi verranno inseriti in uno dei pochi tipi di oggetti speciali come campo. La loro classificazione speciale degli oggetti dipende dal loro comportamento. GraphQL contiene tre tipi di oggetti speciali fondamentali: query, mutazioni e sottoscrizioni. [Per ulteriori informazioni, vedete Oggetti speciali.](#)

Poiché `getAuthor` e `getBook` entrambi richiedono dati, verranno inseriti in un tipo di oggetto Query speciale:

```
type Author {
  authorName: String
  Books: [Book]
}

type Book {
  bookName: String
  Authors: [Author]
}

type Query {
  getAuthor(authorName: String): Author
  getBook(bookName: String): Book
}
```

Le operazioni sono collegate alla query, che a sua volta è collegata allo schema. L'aggiunta di una radice dello schema definirà il tipo di oggetto speciale (Query in questo caso) come uno dei punti di ingresso. Questo può essere fatto usando la schema parola chiave:

```
schema {  
  query: Query  
}  
  
type Author {  
  authorName: String  
  Books: [Book]  
}  
  
type Book {  
  bookName: String  
  Authors: [Author]  
}  
  
type Query {  
  getAuthor(authorName: String): Author  
  getBook(bookName: String): Book  
}
```

Considerando gli ultimi due metodi, `addAuthor` e `addBook` stiamo aggiungendo dati al database, quindi verranno definiti in un tipo di oggetto `Mutation` speciale. Tuttavia, dalla pagina [Tipi](#), sappiamo anche che gli input che fanno riferimento direttamente agli oggetti non sono consentiti perché sono strettamente tipi di output. In questo caso, non possiamo usare `Author` o `Book`, quindi dobbiamo creare un tipo di input con gli stessi campi. In questo esempio, abbiamo aggiunto `AuthorInput` e `BookInput`, entrambi, accettano gli stessi campi dei rispettivi tipi. Quindi, creiamo la nostra mutazione usando gli input come parametri:

```
schema {  
  query: Query  
  mutation: Mutation  
}  
  
type Author {  
  authorName: String  
  Books: [Book]  
}
```



```
input AuthorInput {
  authorName: String!
  Books: [BookInput]
}

type Book {
  bookName: String!
  Authors: [Author]
}

input BookInput {
  bookName: String!
  Authors: [AuthorInput]
}

type Query {
  getAuthor(authorName: String!): Author
  getBook(bookName: String!): Book
}

type Mutation {
  addAuthor(input: [BookInput]!): Author
  addBook(input: [AuthorInput]!): Book
}
```

Rivediamo cosa abbiamo appena fatto:

1. Abbiamo creato uno schema con i `Author` tipi `Book` e per rappresentare le nostre entità.
2. Abbiamo aggiunto i campi contenenti le caratteristiche delle nostre entità.
3. Abbiamo aggiunto una query per recuperare queste informazioni dal database.
4. Abbiamo aggiunto una mutazione per manipolare i dati nel database.
5. Abbiamo aggiunto tipi di input per sostituire i parametri dei nostri oggetti nella mutazione per rispettare le regole di GraphQL.
6. Abbiamo aggiunto la query e la mutazione al nostro schema principale in modo che l'implementazione GraphQL comprenda la posizione del tipo di radice.

Come puoi vedere, il processo di creazione di uno schema richiede molti concetti tratti dalla modellazione dei dati (in particolare dalla modellazione di database) in generale. Si può pensare che lo schema si adatti alla forma dei dati di origine. Serve anche come modello che il resolver

implementerà. Nelle sezioni seguenti, imparerai come creare uno schema utilizzando vari strumenti e AWS servizi supportati.

Note

Gli esempi nelle sezioni seguenti non sono pensati per essere eseguiti in un'applicazione reale. Servono solo a mostrare i comandi in modo da poter creare applicazioni personalizzate.

Creazione di schemi

Lo schema sarà contenuto in un file chiamato `schema.graphql`. AWS AppSync consente agli utenti di creare nuovi schemi per il proprio APIs GraphQL utilizzando vari metodi. In questo esempio, creeremo un'API vuota insieme a uno schema vuoto.

Console

1. Accedi a Console di gestione AWS e apri la [AppSyncconsole](#).
 - a. Nel pannello di controllo, scegliere Create API (Crea API).
 - b. In Opzioni API, scegli GraphQL APIs, Progetta da zero, quindi Avanti.
 - i. Per il nome dell'API, modifica il nome precompilato in base alle esigenze dell'applicazione.
 - ii. Per i dettagli di contatto, puoi inserire un punto di contatto per identificare un gestore dell'API. Questo campo è opzionale.
 - iii. In Configurazione API privata, puoi abilitare le funzionalità dell'API privata. È possibile accedere a un'API privata solo da un endpoint VPC configurato (VPCE). [Per ulteriori informazioni, consulta Privato. APIs](#)

Non è consigliabile abilitare questa funzionalità per questo esempio. Scegli Avanti dopo aver esaminato i dati inseriti.

- c. In Crea un tipo GraphQL, puoi scegliere di creare una tabella DynamoDB da utilizzare come origine dati o saltare questa operazione e farlo in un secondo momento.

Per questo esempio, scegli Create GraphQL resources in un secondo momento. Creeremo una risorsa in una sezione separata.

- d. Controlla i tuoi input, quindi scegli Crea API.
2. Sarai nella dashboard della tua API specifica. Puoi capirlo perché il nome dell'API sarà nella parte superiore della dashboard. In caso contrario, puoi selezionarlo API nella barra laterale, quindi scegliere l'API nella APIs dashboard.
 - Nella barra laterale sotto il nome dell'API, scegli Schema.
3. Nell'editor dello schema, puoi configurare il tuo file `schema.graphql`. Può essere vuoto o pieno di tipi generati da un modello. Sulla destra, c'è la sezione Resolver per allegare i resolver ai campi dello schema. Non esamineremo i resolver in questa sezione.

CLI

Note

Quando utilizzi la CLI, assicurati di disporre delle autorizzazioni corrette per accedere e creare risorse nel servizio. Potresti voler impostare politiche con [privilegi minimi](#) per gli utenti non amministratori che devono accedere al servizio. Per ulteriori informazioni sulle AWS AppSync politiche, consulta Gestione delle [identità](#) e degli accessi per AWS AppSync. Inoltre, ti consigliamo di leggere prima la versione per console se non l'hai già fatto.

1. [Se non l'hai già fatto, installa la AWS CLI, quindi aggiungi la tua configurazione.](#)
2. Crea un oggetto API GraphQL eseguendo il [create-graphql-api](#) comando.

Dovrai digitare due parametri per questo particolare comando:

1. La name della tua API.
2. Il o authentication-type il tipo di credenziali utilizzate per accedere all'API (IAM, OIDC, ecc.).

Note

Altri parametri, ad esempio, Region devono essere configurati, ma di solito vengono utilizzati per impostazione predefinita i valori di configurazione CLI.

Un comando di esempio può avere il seguente aspetto:

```
aws appsync create-graphql-api --name testAPI123 --authentication-type API_KEY
```

Un output verrà restituito nella CLI. Ecco un esempio:

```
{
  "graphqlApi": {
    "xrayEnabled": false,
    "name": "testAPI123",
    "authenticationType": "API_KEY",
    "tags": {},
    "apiId": "abcdefghijklmnopqrstuvwxyz",
    "uris": {
      "GRAPHQL": "https://zyxwvutsrqponmlkjihgfedcba.appsync-api.us-west-2.amazonaws.com/graphql",
      "REALTIME": "wss://zyxwvutsrqponmlkjihgfedcba.appsync-realtime-api.us-west-2.amazonaws.com/graphql"
    },
    "arn": "arn:aws:appsync:us-west-2:107289374856:apis/abcdefghijklmnopqrstuvwxyz"
  }
}
```

3.

 Note

Si tratta di un comando opzionale che accetta uno schema esistente e lo carica sul AWS AppSync servizio utilizzando un blob base-64. Non utilizzeremo questo comando per questo esempio.

Esegui il comando [start-schema-creation](#).

Dovrai digitare due parametri per questo particolare comando:

1. Proviene `api-id` dal passaggio precedente.
2. Lo schema `definition` è un blob binario codificato in base 64.

Un comando di esempio può avere il seguente aspetto:

```
aws appsync start-schema-creation --api-id abcdefghijklmnopqrstuvwxyz --  
definition "aa1111aa-123b-2bb2-c321-12hgg76cc33v"
```

Verrà restituito un output:

```
{  
  "status": "PROCESSING"  
}
```

Questo comando non restituirà l'output finale dopo l'elaborazione. È necessario utilizzare un comando separato per visualizzare il risultato. [get-schema-creation-status](#) Nota che questi due comandi sono asincroni, quindi puoi controllare lo stato dell'output anche mentre lo schema è ancora in fase di creazione.

CDK

Tip

[Prima di utilizzare il CDK, consigliamo di consultare la documentazione ufficiale del CDK insieme al riferimento CDK. AWS AppSync](#)

I passaggi elencati di seguito mostreranno solo un esempio generale dello snippet utilizzato per aggiungere una particolare risorsa. Questa non è pensata per essere una soluzione funzionante nel codice di produzione. Supponiamo inoltre che tu abbia già un'app funzionante.

1. Il punto di partenza per il CDK è leggermente diverso. Idealmente, il `schema.graphql` file dovrebbe essere già stato creato. Devi solo creare un nuovo file con l'estensione del `.graphql` file. Questo può essere un file vuoto.
2. In generale, potrebbe essere necessario aggiungere la direttiva di importazione al servizio che si sta utilizzando. Ad esempio, può seguire i moduli:

```
import * as x from 'x'; # import wildcard as the 'x' keyword from 'x-service'  
import {a, b, ...} from 'c'; # import {specific constructs} from 'c-service'
```

Per aggiungere un'API GraphQL, il file stack deve importare il servizio: AWS AppSync

```
import * as appsync from 'aws-cdk-lib/aws-appsync';
```

Note

Ciò significa che stiamo importando l'intero servizio con la parola chiave. `appsync`. Per utilizzarlo nella tua app, i tuoi AWS AppSync costrutti utilizzeranno il formato. `appsync.construct_name` Ad esempio, se volessimo creare un'API GraphQL, diremmo. `new appsync.GraphqlApi(args_go_here)` Il passaggio seguente illustra questo.

3. L'API GraphQL più semplice includerà una name per l'API e il schema percorso.

```
const add_api = new appsync.GraphqlApi(this, 'API_ID', {  
  name: 'name_of_API_in_console',  
  schema: appsync.SchemaFile.fromAsset(path.join(__dirname,  
    'schema_name.graphql')),  
});
```

Note

Esaminiamo cosa fa questo frammento. Nell'ambito `diapi`, stiamo creando una nuova API GraphQL chiamando. `appsync.GraphqlApi(scope: Construct, id: string, props: GraphqlApiProps)` L'ambito è `this`, che si riferisce all'oggetto corrente. L'id è `API_ID`, che sarà il nome della risorsa dell'API GraphQL al CloudFormation momento della creazione. `GraphqlApiProps` Contiene la name tua API GraphQL e il. schema `schema` Genererà uno schema (`SchemaFile.fromAsset`) cercando il percorso assoluto (`__dirname`) per il `.graphql` file (`schema_name.graphql`). In uno scenario reale, il file di schema si troverà probabilmente all'interno dell'app CDK.

Per utilizzare le modifiche apportate alla tua API GraphQL, dovrai ridistribuire l'app.

Aggiungere tipi agli schemi

Ora che hai aggiunto lo schema, puoi iniziare ad aggiungere sia i tipi di input che quelli di output. Nota che i tipi qui non devono essere usati nel codice reale; sono solo esempi per aiutarti a comprendere il processo.

Per prima cosa, creeremo un tipo di oggetto. Nel codice reale, non è necessario iniziare con questi tipi. Puoi creare qualsiasi tipo desideri in qualsiasi momento purché segua le regole e la sintassi di GraphQL.

Note

Nelle prossime sezioni verrà utilizzato l'editor di schemi, quindi tienilo aperto.

Console

- È possibile creare un tipo di oggetto utilizzando la `type` parola chiave insieme al nome del tipo:

```
type Type_Name_Goes_Here {}
```

All'interno dell'ambito del tipo, puoi aggiungere campi che rappresentano le caratteristiche dell'oggetto:

```
type Type_Name_Goes_Here {  
  # Add fields here  
}
```

Ecco un esempio:

```
type Obj_Type_1 {  
  id: ID!  
  title: String  
  date: AWSDateTime  
}
```

 Note

In questo passaggio, abbiamo aggiunto un tipo di oggetto generico con un `id` campo obbligatorio memorizzato come `ID`, un `title` campo archiviato come `String` e un `date` campo archiviato come `AWSDatetime`. Per visualizzare un elenco di tipi e campi e le relative funzioni, consulta [Schemi](#). Per visualizzare un elenco di scalari e cosa fanno, consulta il riferimento [Type](#).

CLI

 Note

Ti consigliamo di leggere prima la versione per console se non l'hai già fatto.

- È possibile creare un tipo di oggetto eseguendo il [create-type](#) comando.

Dovrai inserire alcuni parametri per questo particolare comando:

1. La `api-id` della tua API.
2. Il `definition`, o il contenuto del tuo tipo. Nell'esempio della console, questo era:

```
type Obj_Type_1 {  
  id: ID!  
  title: String  
  date: AWSDatetime  
}
```

3. Il `format` tuo input. In questo esempio, stiamo usando `SDL`.

Un comando di esempio può essere simile al seguente:

```
aws appsync create-type --api-id abcdefghijklmnopqrstuvwxyz --definition "type  
Obj_Type_1{id: ID! title: String date: AWSDatetime}" --format SDL
```

Un output verrà restituito nella CLI. Ecco un esempio:


```
{
  "type": {
    "definition": "type Obj_Type_1{id: ID! title: String date:
AWSDateTime}",
    "name": "Obj_Type_1",
    "arn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz/types/Obj_Type_1",
    "format": "SDL"
  }
}
```

Note

In questo passaggio, abbiamo aggiunto un tipo di oggetto generico con un `id` campo obbligatorio memorizzato come `ID`, un `title` campo archiviato come `String` e un `date` campo archiviato come `AWSDateTime`. Per visualizzare un elenco di tipi e campi e le relative funzioni, consulta [Schemi](#). Per visualizzare un elenco di scalari e cosa fanno, vedi [Type](#) reference.

Inoltre, potresti aver capito che l'immissione diretta della definizione funziona per i tipi più piccoli, ma non è possibile aggiungere tipi più grandi o multipli. Puoi scegliere di aggiungere tutto in un `.graphql` file e poi [passarlo come input](#).

CDK

Tip

Prima di utilizzare il CDK, consigliamo di consultare la [documentazione ufficiale del CDK insieme AWS AppSync al riferimento CDK](#).

I passaggi elencati di seguito mostreranno solo un esempio generale dello snippet utilizzato per aggiungere una particolare risorsa. Questa non è pensata per essere una soluzione funzionante nel codice di produzione. Supponiamo inoltre che tu abbia già un'app funzionante.

Per aggiungere un tipo, devi aggiungerlo al tuo `.graphql` file. Ad esempio, l'esempio della console era:

```
type Obj_Type_1 {  
  id: ID!  
  title: String  
  date: AWSDateTime  
}
```

Puoi aggiungere i tuoi tipi direttamente allo schema come qualsiasi altro file.

Note

Per utilizzare le modifiche apportate alla tua API GraphQL, dovrai ridistribuire l'app.

Il [tipo di oggetto](#) ha campi di [tipo scalare](#) come stringhe e numeri interi. AWS AppSync consente inoltre di utilizzare tipi scalari avanzati, ad esempio `AWSDateTime` in aggiunta agli scalari GraphQL di base. Inoltre, è obbligatorio qualsiasi campo che termina con un punto esclamativo.

Il tipo ID scalare, in particolare, è un identificatore univoco che può essere uno dei due. `String` `Int` È possibile controllarli nel codice del resolver per l'assegnazione automatica.

Esistono delle somiglianze tra tipi di oggetti speciali come quelli Query «normali», come nell'esempio precedente, in quanto entrambi usano la `type` parola chiave e sono considerati oggetti. Tuttavia, per i tipi di oggetti speciali (`Query`, `Mutation`, and `Subscription`), il loro comportamento è molto diverso perché sono esposti come punti di ingresso per l'API. Si occupano anche più di modellare le operazioni piuttosto che i dati. Per ulteriori informazioni, consulta [I tipi di interrogazione e mutazione](#).

Per quanto riguarda i tipi di oggetti speciali, il passaggio successivo potrebbe essere quello di aggiungerne uno o più per eseguire operazioni sui dati sagomati. In uno scenario reale, ogni schema GraphQL deve avere almeno un tipo di query root per la richiesta dei dati. Puoi pensare alla query come a uno dei punti di ingresso (o endpoint) del tuo server GraphQL. Aggiungiamo una query come esempio.

Console

- Per creare una query, è sufficiente aggiungerla al file di schema come qualsiasi altro tipo. Una query richiederebbe un Query tipo e una voce nella radice come segue:

```
schema {  
  query: Name_of_Query  
}
```

```
type Name_of_Query {  
  # Add field operation here  
}
```

Nota che *Name_of_Query* in un ambiente di produzione verrà semplicemente chiamato Query nella maggior parte dei casi. Si consiglia di mantenerlo a questo valore. All'interno del tipo di query, puoi aggiungere campi. Ogni campo eseguirà un'operazione nella richiesta. Di conseguenza, la maggior parte, se non tutti, questi campi verranno allegati a un resolver. Tuttavia, questo non ci interessa in questa sezione. Per quanto riguarda il formato dell'operazione sul campo, potrebbe essere simile a questo:

```
Name_of_Query(params): Return_Type # version with params  
Name_of_Query: Return_Type # version without params
```

Ecco un esempio:

```
schema {  
  query: Query  
}  
  
type Query {  
  getObj: [Obj_Type_1]  
}  
  
type Obj_Type_1 {  
  id: ID!  
  title: String  
  date: AWSDateTime  
}
```

Note

In questo passaggio, abbiamo aggiunto un Query tipo e lo abbiamo definito nella schema radice. Il nostro Query tipo ha definito un *getObj* campo che restituisce un elenco di *Obj_Type_1* oggetti. Nota che *Obj_Type_1* è l'oggetto del passaggio precedente. Nel codice di produzione, le operazioni sul campo normalmente funzioneranno con dati modellati da oggetti come *Obj_Type_1*. Inoltre, campi

come quelli `getObj` normalmente dispongono di un resolver per eseguire la logica aziendale. Questo sarà trattato in una sezione diversa.

Come nota aggiuntiva, aggiunge AWS AppSync automaticamente una radice dello schema durante le esportazioni, quindi tecnicamente non è necessario aggiungerla direttamente allo schema. Il nostro servizio elaborerà automaticamente gli schemi duplicati. Lo stiamo aggiungendo qui come best practice.

CLI

Note

Ti consigliamo di leggere prima la versione per console se non l'hai già fatto.

1. Crea una schema radice con una query definizione eseguendo il [create-type](#) comando.

Dovrai inserire alcuni parametri per questo particolare comando:

1. La `api-id` della tua API.
2. Il `definition`, o il contenuto del tuo tipo. Nell'esempio della console, questo era:

```
schema {  
  query: Query  
}
```

3. Il `format` tuo input. In questo esempio, stiamo usando `SDL`.

Un comando di esempio può essere simile al seguente:

```
aws appsync create-type --api-id abcdefghijklmnopqrstuvwxyz --definition "schema  
{query: Query}" --format SDL
```

Un output verrà restituito nella CLI. Ecco un esempio:

```
{  
  "type": {  
    "definition": "schema {query: Query}",  
    "name": "schema",
```

```
"arn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnoprstuvwxyz/types/schema",
  "format": "SDL"
}
}
```

Note

Nota che se non hai inserito qualcosa correttamente nel `create-type` comando, puoi aggiornare la radice dello schema (o qualsiasi tipo nello schema) eseguendo il [update-type](#) comando. In questo esempio, cambieremo temporaneamente la radice dello schema per contenere una `subscription` definizione.

Dovrai inserire alcuni parametri per questo particolare comando:

1. La `api-id` della tua API.
2. Il tuo `type-name` tipo. Nell'esempio della console, questo era `schema`.
3. Il `definition`, o il contenuto del tuo tipo. Nell'esempio della console, questo era:

```
schema {
  query: Query
}
```

Lo schema dopo l'aggiunta di un `subscription` sarà simile al seguente:

```
schema {
  query: Query
  subscription: Subscription
}
```

4. Il `format` tuo contributo. In questo esempio, stiamo usando `SDL`.

Un comando di esempio può essere simile al seguente:

```
aws appsync update-type --api-id abcdefghijklmnoprstuvwxyz --type-name
schema --definition "schema {query: Query subscription: Subscription}"
--format SDL
```

Un output verrà restituito nella CLI. Ecco un esempio:

```
{
  "type": {
    "definition": "schema {query: Query subscription: Subscription}",
    "arn": "arn:aws:appsync:us-west-2:107289374856:apis/
    abcdefghijklmnopqrstuvwxyz/types/schema",
    "format": "SDL"
  }
}
```

L'aggiunta di file preformattati continuerà a funzionare in questo esempio.

2. Crea un Query tipo eseguendo il [create-type](#) comando.

Dovrai inserire alcuni parametri per questo particolare comando:

1. La `api-id` della tua API.
2. Il `definition`, o il contenuto del tuo tipo. Nell'esempio della console, questo era:

```
type Query {
  getObj: [Obj_Type_1]
}
```

3. Il `format` tuo input. In questo esempio, stiamo usando `SDL`.

Un comando di esempio può essere simile al seguente:

```
aws appsync create-type --api-id abcdefghijklmnopqrstuvwxyz --definition "type
Query {getObj: [Obj_Type_1]}" --format SDL
```

Un output verrà restituito nella CLI. Ecco un esempio:

```
{
  "type": {
    "definition": "Query {getObj: [Obj_Type_1]}",
    "name": "Query",
    "arn": "arn:aws:appsync:us-west-2:107289374856:apis/
    abcdefghijklmnopqrstuvwxyz/types/Query",
    "format": "SDL"
  }
}
```

```
}
```

Note

In questo passaggio, abbiamo aggiunto un Query tipo e lo abbiamo definito nella schema radice. Il nostro Query tipo ha definito un `getObj` campo che ha restituito un elenco di `Obj_Type_1` oggetti.

Nel codice schema `principalequery: Query`, la `query:` parte indica che una query è stata definita nello schema, mentre la Query parte indica il nome effettivo dell'oggetto speciale.

CDK

Tip

Prima di utilizzare il CDK, consigliamo di consultare la [documentazione ufficiale del CDK insieme AWS AppSync al riferimento CDK](#).

I passaggi elencati di seguito mostreranno solo un esempio generale dello snippet utilizzato per aggiungere una particolare risorsa. Questa non è pensata per essere una soluzione funzionante nel codice di produzione. Supponiamo inoltre che tu abbia già un'app funzionante.

Dovrai aggiungere la tua query e la radice dello schema al `.graphql` file. Il nostro esempio assomigliava all'esempio seguente, ma ti consigliamo di sostituirlo con il codice dello schema effettivo:

```
schema {  
  query: Query  
}  
  
type Query {  
  getObj: [Obj_Type_1]  
}  
  
type Obj_Type_1 {  
  id: ID!  
  title: String
```

```
date: AWSDateTime
}
```

Puoi aggiungere i tuoi tipi direttamente allo schema come qualsiasi altro file.

Note

L'aggiornamento della radice dello schema è facoltativo. L'abbiamo aggiunta a questo esempio come procedura consigliata.

Per utilizzare le modifiche apportate alla tua API GraphQL, dovrai ridistribuire l'app.

Ora hai visto un esempio di creazione sia di oggetti che di oggetti speciali (query). Hai anche visto come questi possono essere interconnessi per descrivere dati e operazioni. Puoi avere schemi con solo la descrizione dei dati e una o più interrogazioni. Tuttavia, vorremmo aggiungere un'altra operazione per aggiungere dati all'origine dati. Aggiungeremo un altro tipo di oggetto speciale chiamato `Mutation` che modifica i dati.

Console

- Verrà chiamata una mutazione. `Mutation` Ad esempio `Query`, le operazioni sul campo interne `Mutation` descriveranno un'operazione e saranno collegate a un resolver. Inoltre, nota che dobbiamo definirlo nella schema radice perché è un tipo di oggetto speciale. Ecco un esempio di mutazione:

```
schema {
  mutation: Name_of_Mutation
}

type Name_of_Mutation {
  # Add field operation here
}
```

Una mutazione tipica verrà elencata nella radice come una query. La mutazione viene definita utilizzando la `type` parola chiave insieme al nome. `Name_of_Mutation` di solito viene chiamato `Mutation`, quindi consigliamo di mantenerlo in questo modo. Ogni campo eseguirà anche un'operazione. Per quanto riguarda il formato dell'operazione sul campo, potrebbe essere simile a questo:


```
Name_of_Mutation(params): Return_Type # version with params
Name_of_Mutation: Return_Type # version without params
```

Ecco un esempio:

```
schema {
  query: Query
  mutation: Mutation
}

type Obj_Type_1 {
  id: ID!
  title: String
  date: AWSDateTime
}

type Query {
  getObj: [Obj_Type_1]
}

type Mutation {
  addObj(id: ID!, title: String, date: AWSDateTime): Obj_Type_1
}
```

Note

In questo passaggio, abbiamo aggiunto un `Mutation` tipo con un `addObj` campo. Riassumiamo cosa fa questo campo:

```
addObj(id: ID!, title: String, date: AWSDateTime): Obj_Type_1
```

`addObj` sta usando l'`Obj_Type_1` oggetto per eseguire un'operazione. Ciò è evidente a causa dei campi, ma la sintassi lo dimostra nel tipo : `Obj_Type_1` restituito. All'interno `addObj`, accetta i `id` `date` campi e e dell'`Obj_Type_1` oggetto come parametri. `title` Come puoi vedere, assomiglia molto a una dichiarazione di metodo. Tuttavia, non abbiamo ancora descritto il comportamento del nostro metodo. Come affermato in precedenza, lo schema serve solo a definire quali saranno i dati e le operazioni e non come funzioneranno. L'implementazione dell'effettiva logica aziendale avverrà più avanti, quando creeremo i nostri primi resolver.

Una volta che hai finito con lo schema, c'è un'opzione per esportarlo come file. `schema.graphql` Nell'editor dello schema, puoi scegliere **Esporta schema** per scaricare il file in un formato supportato.

Come nota aggiuntiva, aggiunge AWS AppSync automaticamente una radice dello schema durante le esportazioni, quindi tecnicamente non è necessario aggiungerla direttamente allo schema. Il nostro servizio elaborerà automaticamente gli schemi duplicati. Lo stiamo aggiungendo qui come best practice.

CLI

Note

Ti consigliamo di leggere prima la versione per console se non l'hai già fatto.

1. Aggiorna lo schema root eseguendo il [update-type](#) comando.

Dovrai inserire alcuni parametri per questo particolare comando:

1. La `api-id` della tua API.
2. Il tuo `type-name` tipo. Nell'esempio della console, questo era `schema`.
3. Il `definition`, o il contenuto del tuo tipo. Nell'esempio della console, questo era:

```
schema {  
  query: Query  
  mutation: Mutation  
}
```

4. Il `format` tuo input. In questo esempio, stiamo usando `SDL`.

Un comando di esempio può essere simile al seguente:

```
aws appsync update-type --api-id abcdefghijklmnopqrstuvwxyz --type-name schema  
--definition "schema {query: Query mutation: Mutation}" --format SDL
```

Un output verrà restituito nella CLI. Ecco un esempio:

```
{
  "type": {
    "definition": "schema {query: Query mutation: Mutation}",
    "arn": "arn:aws:appsync:us-west-2:107289374856:apis/
    abcdefghijklmnopqrstuvwxyz/types/schema",
    "format": "SDL"
  }
}
```

2. Crea un Mutation tipo eseguendo il [create-type](#) comando.

Dovrai inserire alcuni parametri per questo particolare comando:

1. La `api-id` della tua API.
2. Il `definition`, o il contenuto del tuo tipo. Nell'esempio della console, questo era

```
type Mutation {
  addObj(id: ID!, title: String, date: AWSDateTime): Obj_Type_1
}
```

3. Il `format` tuo input. In questo esempio, stiamo usando `SDL`.

Un comando di esempio può essere simile al seguente:

```
aws appsync create-type --api-id abcdefghijklmnopqrstuvwxyz --definition "type
Mutation {addObj(id: ID! title: String date: AWSDateTime): Obj_Type_1}" --
format SDL
```

Un output verrà restituito nella CLI. Ecco un esempio:

```
{
  "type": {
    "definition": "type Mutation {addObj(id: ID! title: String date:
    AWSDateTime): Obj_Type_1}",
    "name": "Mutation",
    "arn": "arn:aws:appsync:us-west-2:107289374856:apis/
    abcdefghijklmnopqrstuvwxyz/types/Mutation",
    "format": "SDL"
  }
}
```

CDK

Tip

[Prima di utilizzare il CDK, consigliamo di consultare la documentazione ufficiale del CDK insieme AWS AppSync al riferimento CDK.](#)

I passaggi elencati di seguito mostreranno solo un esempio generale dello snippet utilizzato per aggiungere una particolare risorsa. Questa non è pensata per essere una soluzione funzionante nel codice di produzione. Supponiamo inoltre che tu abbia già un'app funzionante.

Dovrai aggiungere la tua query e la radice dello schema al `.graphql` file. Il nostro esempio assomigliava all'esempio seguente, ma ti consigliamo di sostituirlo con il codice dello schema effettivo:

```
schema {  
  query: Query  
  mutation: Mutation  
}  
  
type Obj_Type_1 {  
  id: ID!  
  title: String  
  date: AWSDateTime  
}  
  
type Query {  
  getObj: [Obj_Type_1]  
}  
  
type Mutation {  
  addObj(id: ID!, title: String, date: AWSDateTime): Obj_Type_1  
}
```

Note

L'aggiornamento della radice dello schema è facoltativo. L'abbiamo aggiunta a questo esempio come procedura consigliata.

Per utilizzare le modifiche apportate alla tua API GraphQL, dovrai ridistribuire l'app.

Considerazioni facoltative: utilizzo delle enumerazioni come stati

A questo punto, sapete come creare uno schema di base. Tuttavia, ci sono molte cose che è possibile aggiungere per aumentare la funzionalità dello schema. Una caratteristica comune nelle applicazioni è l'uso di enumerazioni come stati. È possibile utilizzare un enum per forzare un valore specifico da un insieme di valori da scegliere quando viene chiamato. Questo è utile per cose che sai non cambieranno drasticamente per lunghi periodi di tempo. Ipoteticamente parlando, potremmo aggiungere un enum che restituisca il codice di stato o la stringa nella risposta.

Ad esempio, supponiamo di creare un'app per social media che memorizza i dati dei post di un utente nel backend. Il nostro schema contiene un `Post` tipo che rappresenta i dati di un singolo post:

```
type Post {  
  id: ID!  
  title: String  
  date: AWSDateTime  
  poststatus: PostStatus  
}
```

Il nostro `Post` conterrà un unico `id` `post title` `date` di pubblicazione e un enum chiamato `PostStatus` che rappresenta lo stato del post così come viene elaborato dall'app. Per le nostre operazioni, avremo una query che restituisce tutti i dati dei post:

```
type Query {  
  getPosts: [Post]  
}
```

Avremo anche una mutazione che aggiunge post alla fonte di dati:

```
type Mutation {  
  addPost(id: ID!, title: String, date: AWSDateTime, poststatus: PostStatus): Post  
}
```

Guardando il nostro schema, l'`PostStatus` enum potrebbe avere diversi stati. Potremmo volere che i tre stati di base si chiamino `success` (post elaborato con successo), `pending` (post in fase di

elaborazione) e `error` (post non in grado di essere elaborato). Per aggiungere l'enum, potremmo fare questo:

```
enum PostStatus {  
  success  
  pending  
  error  
}
```

Lo schema completo potrebbe essere simile a questo:

```
schema {  
  query: Query  
  mutation: Mutation  
}  
  
type Post {  
  id: ID!  
  title: String  
  date: AWSDateTime  
  poststatus: PostStatus  
}  
  
type Mutation {  
  addPost(id: ID!, title: String, date: AWSDateTime, poststatus: PostStatus): Post  
}  
  
type Query {  
  getPosts: [Post]  
}  
  
enum PostStatus {  
  success  
  pending  
  error  
}
```

Se un utente aggiunge un `Post` nell'applicazione, l'`addPost` operazione verrà chiamata per elaborare quei dati. Man mano che il resolver collegato `addPost` elabora i dati, li aggiornerà continuamente `poststatus` con lo stato dell'operazione. Quando richiesto, `Post` conterrà lo stato finale dei dati. Tieni presente che stiamo solo descrivendo come vogliamo che i dati funzionino nello schema.

Stiamo dando molte supposizioni sull'implementazione dei nostri resolver, che implementeranno l'effettiva logica aziendale per la gestione dei dati per soddisfare la richiesta.

Considerazioni opzionali - Abbonamenti

Le sottoscrizioni in AWS AppSync vengono richiamate come risposta a una mutazione. È possibile configurare ciò con un tipo `Subscription` e una direttiva `@aws_subscribe()` nello schema, per indicare quali mutazioni richiamano una o più sottoscrizioni. [Per ulteriori informazioni sulla configurazione degli abbonamenti, consulta Dati in tempo reale.](#)

Considerazioni facoltative: relazioni e impaginazione

Supponiamo di avere un milione `Posts` archiviato in una tabella `DynamoDB` e di voler restituire alcuni di quei dati. Tuttavia, la query di esempio riportata sopra restituisce solo tutti i post. Non vorrai recuperarli tutti ogni volta che fai una richiesta. Invece, dovresti [sfogliarli per pagina](#). Apporta le modifiche seguenti allo schema:

- Nel `getPosts` campo, aggiungi due argomenti di input: `nextToken` (iteratore) e `limit` (limite di iterazione).
- Aggiungi un nuovo `PostIterator` tipo contenente `Posts` (recupera l'elenco degli `Post` oggetti) e `nextToken` (iteratore) i campi.
- Modifica `getPosts` in modo che restituisca `PostIterator` e non un elenco di `Post` oggetti.

```
schema {  
  query: Query  
  mutation: Mutation  
}  
  
type Post {  
  id: ID!  
  title: String  
  date: AWSDateTime  
  poststatus: PostStatus  
}  
  
type Mutation {  
  addPost(id: ID!, title: String, date: AWSDateTime, poststatus: PostStatus): Post  
}
```

```
type Query {  
  getPosts(limit: Int, nextToken: String): PostIterator  
}  
  
enum PostStatus {  
  success  
  pending  
  error  
}  
  
type PostIterator {  
  posts: [Post]  
  nextToken: String  
}
```

Il `PostIterator` tipo consente di restituire una parte dell'elenco di `Post` oggetti e `nextToken` di ottenere la parte successiva. All'interno `PostIterator`, c'è un elenco di `Post` elementi (`[Post]`) che viene restituito con un token di paginazione (`nextToken`). Nel AWS AppSync, questo verrebbe collegato ad Amazon DynamoDB tramite un resolver e generato automaticamente come token crittografato. Il modello converte il valore dell'argomento `limit` nel parametro `maxResults` e dell'argomento `nextToken` nel parametro `exclusiveStartKey`. Per alcuni esempi e gli esempi di modelli incorporati nella AWS AppSync console, consulta [Resolver](#) reference (). JavaScript

Collegamento di un'origine dati in AWS AppSync

Le fonti di dati sono risorse del tuo AWS account con cui GraphQL APIs può interagire. AWS AppSync supporta una moltitudine di fonti di dati come Amazon DynamoDB AWS Lambda, database relazionali (Amazon Aurora Serverless) OpenSearch , Amazon Service ed endpoint HTTP. Un' AWS AppSync API può essere configurata per interagire con più fonti di dati, consentendoti di aggregare i dati in un'unica posizione. AWS AppSync puoi utilizzare AWS le risorse esistenti del tuo account o effettuare il provisioning di tabelle DynamoDB per tuo conto a partire da una definizione di schema.

La sezione seguente ti mostrerà come collegare un'origine dati all'API GraphQL.

Tipi di fonti di dati

Ora che hai creato uno schema nella AWS AppSync console, puoi allegare un'origine dati ad esso. Quando crei inizialmente un'API, è possibile effettuare il provisioning di una tabella Amazon DynamoDB durante la creazione dello schema predefinito. Tuttavia, non tratteremo questa opzione in questa sezione. Puoi vederne un esempio nella sezione [Avvio di uno schema](#).

Invece, esamineremo tutte le fonti di dati AWS AppSync supportate. Ci sono molti fattori che contribuiscono alla scelta della soluzione giusta per la propria applicazione. Le sezioni seguenti forniranno un contesto aggiuntivo per ciascuna fonte di dati. Per informazioni generali sulle fonti di dati, consulta [Fonti di dati](#).

Amazon DynamoDB

Amazon DynamoDB è una delle principali soluzioni AWS di storage per applicazioni scalabili. Il componente principale di DynamoDB è la tabella, che è semplicemente una raccolta di dati. In genere creerai tabelle basate su entità come Book o Author. Le informazioni sulle voci della tabella vengono memorizzate come elementi, ovvero gruppi di campi univoci per ogni voce. Un elemento completo rappresenta un elemento row/record nel database. Ad esempio, un elemento di una Book voce potrebbe includere title e author insieme ai relativi valori. I singoli campi come title e author sono chiamati attributi, che sono simili ai valori delle colonne nei database relazionali.

Come puoi immaginare, le tabelle verranno utilizzate per archiviare i dati dell'applicazione. AWS AppSync consente di collegare le tabelle DynamoDB all'API GraphQL per manipolare i dati. Prendete questo [caso d'uso dal blog](#) Front-end per web e dispositivi mobili. Questa applicazione consente agli utenti di iscriversi a un'app di social media. Gli utenti possono unirsi a gruppi e caricare post che vengono trasmessi ad altri utenti iscritti al gruppo. La loro applicazione archivia le informazioni su utenti, post e gruppi di utenti in DynamoDB. L'API GraphQL (gestita da AWS AppSync) si interfaccia con la tabella DynamoDB. Quando un utente apporta una modifica al sistema che si rifletterà sul front-end, l'API GraphQL recupera queste modifiche e le trasmette ad altri utenti in tempo reale.

AWS Lambda

Lambda è un servizio basato sugli eventi che crea automaticamente le risorse necessarie per eseguire il codice in risposta a un evento. Lambda utilizza le funzioni, che sono istruzioni di gruppo contenenti il codice, le dipendenze e le configurazioni per l'esecuzione di una risorsa. Le funzioni vengono eseguite automaticamente quando rilevano un trigger, un gruppo di attività che richiamano la funzione. Un trigger può essere qualcosa come un'applicazione che effettua una chiamata API, un AWS servizio del tuo account che attiva una risorsa, ecc. Quando attivate, le funzioni elaboreranno gli eventi, che sono documenti JSON contenenti i dati da modificare.

Lambda è utile per eseguire codice senza dover fornire le risorse per eseguirlo. Prendi questo [caso d'uso](#) dal blog Web e mobile di Front-end. Questo caso d'uso è un po' simile a quello illustrato nella sezione DynamoDB. In questa applicazione, l'API GraphQL è responsabile della definizione delle operazioni per cose come l'aggiunta di post (mutazioni) e il recupero di tali dati (query). Per implementare la funzionalità delle loro operazioni (ad esempio, `getPostsByAuthor` (`author:`

`String ! ) : [ Post ])getPost ( id: String ! ) : Post`, utilizzano le funzioni Lambda per elaborare le richieste in entrata. Nell'opzione 2: AWS AppSync con il resolver Lambda, utilizzano il AWS AppSync servizio per mantenere lo schema e collegare un'origine dati Lambda a una delle operazioni. Quando viene chiamata l'operazione, Lambda si interfaccia con il proxy Amazon RDS per eseguire la logica di business sul database.

Amazon RDS

Amazon RDS consente di creare e configurare rapidamente database relazionali. In Amazon RDS, creerai un'istanza di database generica che fungerà da ambiente di database isolato nel cloud. In questo caso, utilizzerai un motore DB, che è il software RDBMS effettivo (PostgreSQL, MySQL, ecc.). Il servizio alleggerisce gran parte del lavoro di backend fornendo scalabilità tramite AWS l'infrastruttura, servizi di sicurezza come patch e crittografia e costi amministrativi ridotti per le implementazioni.

Prendiamo lo stesso [caso d'uso](#) dalla sezione Lambda. Nell'opzione 3: AWS AppSync con Amazon RDS resolver, un'altra opzione presentata è il collegamento diretto dell'API AWS AppSync GraphQL ad Amazon RDS. Utilizzando un'[API di dati](#), associano il database all'API GraphQL. Un resolver è collegato a un campo (di solito una query, una mutazione o una sottoscrizione) e implementa le istruzioni SQL necessarie per accedere al database. Quando il client effettua una richiesta che richiama il campo, il resolver esegue le istruzioni e restituisce la risposta.

Amazon EventBridge

In EventBridge, creerai bus di eventi, ossia pipeline che ricevono eventi dai servizi o dalle applicazioni che colleghi (la fonte dell'evento) e li elaborano in base a una serie di regole. Un evento è un cambiamento di stato in un ambiente di esecuzione, mentre una regola è un insieme di filtri per gli eventi. Una regola segue uno schema di evento o i metadati della modifica dello stato di un evento (id, regione, numero di account, ARN, ecc.). Quando un evento corrisponde al modello dell'evento, EventBridge invierà l'evento attraverso la pipeline al servizio di destinazione (target) e attiverà l'azione specificata nella regola.

EventBridge è utile per indirizzare le operazioni di modifica dello stato verso altri servizi. Prendi questo [caso d'uso](#) dal blog Web e mobile di Front-end. L'esempio illustra una soluzione di e-commerce che dispone di diversi team che gestiscono servizi diversi. Uno di questi servizi fornisce aggiornamenti sugli ordini al cliente in ogni fase della consegna (ordine effettuato, in corso, spedito, consegnato, ecc.) sul front-end. Tuttavia, il team di front-end che gestisce questo servizio non ha accesso diretto ai dati del sistema di ordinazione, poiché sono gestiti da un team di backend separato. Il sistema di ordinazione del team di backend è anche descritto come una scatola nera,

quindi è difficile raccogliere informazioni sul modo in cui strutturano i dati. Tuttavia, il team di backend ha creato un sistema che pubblicava i dati degli ordini tramite un bus di eventi gestito da EventBridge. Per accedere ai dati provenienti dal bus degli eventi e indirizzarli al front-end, il team del front-end ha creato un nuovo target che puntava alla loro API GraphQL installata. AWS AppSync Inoltre, hanno creato una regola per inviare solo i dati relativi all'aggiornamento dell'ordine. Quando viene effettuato un aggiornamento, i dati del bus degli eventi vengono inviati all'API GraphQL. Lo schema nell'API elabora i dati, quindi li passa al front-end.

Nessuna fonte di dati

Se non hai intenzione di utilizzare un'origine dati, puoi impostarla su `null`. Una fonte di `null` dati, sebbene sia ancora esplicitamente classificata come fonte di dati, non è un supporto di archiviazione. In genere, un resolver richiamerà una o più fonti di dati a un certo punto per elaborare la richiesta. Tuttavia, ci sono situazioni in cui potrebbe non essere necessario manipolare una fonte di dati. L'impostazione dell'origine dati su `null` eseguirà la richiesta, salterà la fase di invocazione dei dati, quindi eseguirà la risposta.

Prendiamo lo stesso [caso d'uso](#) della sezione EventBridge. Nello schema, la mutazione elabora l'aggiornamento dello stato, quindi lo invia agli abbonati. Ricordando come funzionano i resolver, di solito c'è almeno una chiamata alla fonte di dati. Tuttavia, i dati in questo scenario sono già stati inviati automaticamente dal bus degli eventi. Ciò significa che non è necessario che la mutazione esegua una chiamata alla fonte di dati; lo stato dell'ordine può essere semplicemente gestito localmente. La mutazione è impostata su `null`, che funge da valore pass-through senza richiamare l'origine dei dati. Lo schema viene quindi popolato con i dati, che vengono inviati agli abbonati.

OpenSearch

Amazon OpenSearch Service è una suite di strumenti per implementare la ricerca di testo completo, la visualizzazione dei dati e la registrazione. Puoi utilizzare questo servizio per interrogare i dati strutturati che hai caricato.

In questo servizio, creerai istanze di OpenSearch. Questi sono chiamati nodi. In un nodo, aggiungerai almeno un indice. Gli indici sono concettualmente un po' come le tabelle nei database relazionali. (Tuttavia, OpenSearch non è conforme ad ACID, quindi non dovrebbe essere usato in questo modo). Compilerai il tuo indice con i dati che caricherai sul servizio. OpenSearch Una volta caricati, i dati verranno indicizzati in uno o più shard presenti nell'indice. Uno shard è come una partizione dell'indice che contiene alcuni dati e può essere interrogato separatamente dagli altri shard. Una volta caricati, i dati saranno strutturati come file JSON chiamati documenti. È quindi possibile interrogare il nodo per i dati nel documento.

Endpoint HTTP

È possibile utilizzare gli endpoint HTTP come fonti di dati. AWS AppSync può inviare richieste agli endpoint con le informazioni pertinenti come parametri e payload. La risposta HTTP verrà esposta al resolver, che restituirà la risposta finale al termine delle sue operazioni.

Aggiungere una fonte di dati

Se hai creato un'origine dati, puoi collegarla al AWS AppSync servizio e, più specificamente, all'API.

Console

1. Accedi a Console di gestione AWS e apri la [AppSyncconsole](#).
 - a. Scegli la tua API nella dashboard.
 - b. Nella barra laterale, scegli Fonti dati.
2. Seleziona Create data source (Crea origine dati).
 - a. Assegna un nome alla tua fonte di dati. Puoi anche dargli una descrizione, ma è facoltativa.
 - b. Scegli il tipo di origine dati.
 - c. Per DynamoDB, dovrai scegliere la tua regione, quindi la tabella nella regione. Puoi dettare le regole di interazione con la tabella scegliendo di creare un nuovo ruolo generico nella tabella o importando un ruolo esistente per la tabella. È possibile abilitare il [controllo delle versioni](#), che consente di creare automaticamente versioni dei dati per ogni richiesta quando più client tentano di aggiornare i dati contemporaneamente. Il controllo delle versioni viene utilizzato per conservare e gestire più varianti di dati per scopi di rilevamento e risoluzione dei conflitti. Puoi anche abilitare la generazione automatica dello schema, che prende la tua fonte di dati e genera parte del CRUDList, e Query le operazioni necessarie per accedervi nello schema.

Infatti OpenSearch, dovrai scegliere la tua regione, quindi il dominio (cluster) nella regione. Puoi dettare le regole di interazione con il tuo dominio scegliendo di creare un nuovo ruolo di tabella generico o importando un ruolo esistente per la tabella.

Per Lambda, dovrai scegliere la tua regione, quindi l'ARN della funzione Lambda nella regione. Puoi dettare le regole di interazione con la tua funzione Lambda scegliendo di creare un nuovo ruolo generico nella tabella o importando un ruolo esistente per la tabella.

Per HTTP, dovrai inserire il tuo endpoint HTTP.

Perché EventBridge, dovrai scegliere la tua regione, quindi l'autobus dell'evento nella regione. Puoi dettare le regole di interazione con il tuo event bus scegliendo di creare un nuovo ruolo generico nella tabella o importando un ruolo esistente per la tabella.

Per RDS, dovrai scegliere la tua regione, quindi l'archivio segreto (nome utente e password), il nome del database e lo schema.

Per nessuno, aggiungerai un'origine dati senza un'origine dati effettiva. Questo serve per gestire i resolver localmente anziché tramite una fonte di dati effettiva.

 Note

Se stai importando ruoli esistenti, hanno bisogno di una politica di fiducia. Per ulteriori informazioni, consulta la [policy di fiducia di IAM](#).

3. Scegli Create (Crea).

 Note

In alternativa, se stai creando un'origine dati DynamoDB, puoi andare alla pagina Schema nella console, scegliere Crea risorse nella parte superiore della pagina, quindi compilare un modello predefinito da convertire in tabella. In questa opzione, compilerai o importerai il tipo di base, configurerai i dati di base della tabella, inclusa la chiave di partizione, ed esaminerai le modifiche allo schema.

CLI

- Crea la tua fonte di dati eseguendo il [create-data-source](#) comando.

Dovrai inserire alcuni parametri per questo particolare comando:

1. La `api-id` della tua API.
2. Il nome tuo tavolo.
3. La fonte `type` dei dati. A seconda del tipo di origine dati scelto, potrebbe essere necessario inserire un tag `service-role-arn` e un `-config` tag.

Un comando di esempio può avere il seguente aspetto:

```
aws appsync create-data-source --api-id abcdefghijklmnopqrstuvwxyz  
--name data_source_name --type data_source_type --service-role-arn  
arn:aws:iam::107289374856:role/role_name --[data_source_type]-config {params}
```

CDK

Tip

Prima di utilizzare il CDK, consigliamo di consultare la [documentazione ufficiale del CDK insieme AWS AppSync al riferimento CDK](#).

I passaggi elencati di seguito mostreranno solo un esempio generale dello snippet utilizzato per aggiungere una particolare risorsa. Questa non è pensata per essere una soluzione funzionante nel codice di produzione. Supponiamo inoltre che tu abbia già un'app funzionante.

Per aggiungere una particolare fonte di dati, dovrai aggiungere il costrutto al tuo file stack. Un elenco di tipi di fonti di dati è disponibile qui:

- [DynamoDbDataSource](#)
- [EventBridgeDataSource](#)
- [HttpDataSource](#)
- [LambdaDataSource](#)
- [NoneDataSource](#)
- [OpenSearchDataSource](#)
- [RdsDataSource](#)

1. In generale, potresti dover aggiungere la direttiva di importazione al servizio che stai utilizzando. Ad esempio, può seguire i moduli:

```
import * as x from 'x'; # import wildcard as the 'x' keyword from 'x-service'  
import {a, b, ...} from 'c'; # import {specific constructs} from 'c-service'
```

Ad esempio, ecco come importare i servizi AWS AppSync e DynamoDB:

```
import * as appsync from 'aws-cdk-lib/aws-appsync';
import * as dynamodb from 'aws-cdk-lib/aws-dynamodb';
```

2. Alcuni servizi come RDS richiedono una configurazione aggiuntiva nel file stack prima di creare l'origine dati (ad esempio, creazione di VPC, ruoli e credenziali di accesso). Consulta gli esempi nelle pagine CDK pertinenti per ulteriori informazioni.
3. Per la maggior parte delle fonti di dati, in particolare AWS i servizi, creerai una nuova istanza dell'origine dati nel tuo file stack. In genere, avrà il seguente aspetto:

```
const add_data_source_func = new service_scope.resource_name(scope: Construct,
  id: string, props: data_source_props);
```

Ad esempio, ecco un esempio di tabella Amazon DynamoDB:

```
const add_ddb_table = new dynamodb.Table(this, 'Table_ID', {
  partitionKey: {
    name: 'id',
    type: dynamodb.AttributeType.STRING,
  },
  sortKey: {
    name: 'id',
    type: dynamodb.AttributeType.STRING,
  },
  tableClass: dynamodb.TableClass.STANDARD,
});
```

Note

La maggior parte delle fonti di dati avrà almeno una proprietà obbligatoria (sarà indicata senza un simbolo). ? Consultate la documentazione del CDK per vedere quali oggetti di scena sono necessari.

4. Successivamente, è necessario collegare l'origine dati all'API GraphQL. Il metodo consigliato è aggiungerlo quando crei una funzione per il tuo risolutore di pipeline. Ad esempio, lo snippet seguente è una funzione che analizza tutti gli elementi in una tabella DynamoDB:

```
const add_func = new appsync.AppsyncFunction(this, 'func_ID', {
  name: 'func_name_in_console',
  add_api,
  dataSource: add_api.addDynamoDbDataSource('data_source_name_in_console',
  add_ddb_table),
  code: appsync.Code.fromInline(`
    export function request(ctx) {
      return { operation: 'Scan' };
    }

    export function response(ctx) {
      return ctx.result.items;
    }
  `),
  runtime: appsync.FunctionRuntime.JS_1_0_0,
});
```

Nelle `dataSource` proprietà, puoi chiamare l'API GraphQL `add_api()` e utilizzare uno dei suoi metodi integrati `addDynamoDbDataSource()` per creare l'associazione tra la tabella e l'API GraphQL. Gli argomenti sono il nome di questo collegamento che esisterà nella AWS AppSync console (`data_source_name_in_console` in questo esempio) e il metodo della tabella (`add_ddb_table`). Ulteriori informazioni su questo argomento verranno rivelate nella prossima sezione quando inizierai a creare resolver.

Esistono metodi alternativi per collegare una fonte di dati. Tecnicamente è possibile aggiungerlo `api` all'elenco degli oggetti di scena nella funzione di tabella. Ad esempio, ecco lo snippet del passaggio 3 ma con un oggetto di `api` scena contenente un'API GraphQL:

```
const add_api = new appsync.GraphqlApi(this, 'API_ID', {
  ...
});

const add_ddb_table = new dynamodb.Table(this, 'Table_ID', {
  ...

  api: add_api
});
```

In alternativa, puoi chiamare il costrutto separatamente: `GraphqlApi`


```
const add_api = new appsync.GraphqlApi(this, 'API_ID', {
    ...
});

const add_ddb_table = new dynamodb.Table(this, 'Table_ID', {
    ...
});

const link_data_source =
    add_api.addDynamoDbDataSource('data_source_name_in_console', add_ddb_table);
```

Consigliamo di creare l'associazione solo negli oggetti di scena della funzione. Altrimenti, dovrai collegare manualmente la funzione del resolver alla fonte di dati nella AWS AppSync console (se vuoi continuare a utilizzare il valore della console `data_source_name_in_console`) o creare un'associazione separata nella funzione con un altro nome come `data_source_name_in_console_2`. Ciò è dovuto alle limitazioni nel modo in cui gli oggetti di scena elaborano le informazioni.

Note

Dovrai ridistribuire l'app per vedere le modifiche.

Politica di fiducia IAM

Se utilizzi un ruolo IAM esistente per la tua origine dati, devi concedere a quel ruolo le autorizzazioni appropriate per eseguire operazioni sulla tua AWS risorsa, ad esempio `PutItem` su una tabella Amazon DynamoDB. È inoltre necessario modificare la policy di fiducia su quel ruolo per consentirne AWS AppSync l'utilizzo per l'accesso alle risorse, come illustrato nella seguente policy di esempio:

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "appsync.amazonaws.com"
```

```
    },  
    "Action": "sts:AssumeRole"  
  }  
]  
}
```

Puoi anche aggiungere condizioni alla tua politica di fiducia per limitare l'accesso all'origine dati, se lo desideri. Attualmente, `SourceArn` e `SourceAccount` chiavi possono essere utilizzate in queste condizioni. Ad esempio, la seguente politica limita l'accesso all'origine dati all'`account123456789012`:

JSON

```
{  
  "Version": "2012-10-17",  
  "Statement": [  
    {  
      "Effect": "Allow",  
      "Principal": {  
        "Service": "appsync.amazonaws.com"  
      },  
      "Action": "sts:AssumeRole",  
      "Condition": {  
        "StringEquals": {  
          "aws:SourceAccount": "123456789012"  
        }  
      }  
    }  
  ]  
}
```

In alternativa, puoi limitare l'accesso a un'origine dati a un'API specifica `abcdefghijklmnopq`, ad esempio utilizzando la seguente politica:

JSON

```
{  
  "Version": "2012-10-17",  
  "Statement": [  
    {  
      "Effect": "Allow",  
      "Principal": {  
        "Service": "appsync.amazonaws.com"  
      },  
      "Action": "sts:AssumeRole",  
      "Condition": {  
        "StringEquals": {  
          "aws:SourceAccount": "123456789012"  
        }  
      }  
    }  
  ]  
}
```

```
{
  "Effect": "Allow",
  "Principal": {
    "Service": "appsync.amazonaws.com"
  },
  "Action": "sts:AssumeRole",
  "Condition": {
    "ArnEquals": {
      "aws:SourceArn": "arn:aws:appsync:us-west-2:123456789012:apis/
abcdefghijklmnopq"
    }
  }
}
```

Puoi limitare l'accesso a tutti AWS AppSync APIs da una regione specifica `us-east-1`, ad esempio utilizzando la seguente politica:

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "appsync.amazonaws.com"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "ArnEquals": {
          "aws:SourceArn": "arn:aws:appsync:us-east-1:123456789012:apis/*"
        }
      }
    }
  ]
}
```

Nella prossima sezione ([Configurazione dei resolver](#)), aggiungeremo la nostra logica aziendale del resolver e la collegheremo ai campi del nostro schema per elaborare i dati nella nostra fonte di dati.

Per ulteriori informazioni sulla configurazione delle politiche relative ai ruoli, consulta [Modificare un ruolo](#) nella Guida per l'utente IAM.

[Per ulteriori informazioni sull'accesso tra account ai resolver per AWS AppSync, consulta AWS Lambda Creazione di resolver tra account per. AWS LambdaAWS AppSync](#)

Configurazione dei resolver in AWS AppSync

Nelle sezioni precedenti, hai imparato a creare lo schema e l'origine dati GraphQL, quindi a collegarli insieme nel AWS AppSync servizio. Nel vostro schema, potreste aver stabilito uno o più campi (operazioni) nella query e nella mutazione. Sebbene lo schema descrivesse i tipi di dati che le operazioni avrebbero richiesto alla fonte dei dati, non ha mai implementato il modo in cui tali operazioni si sarebbero comportate rispetto ai dati.

Il comportamento di un'operazione è sempre implementato nel resolver, che sarà collegato al campo che esegue l'operazione. [Per ulteriori informazioni sul funzionamento dei resolver in generale, consulta la pagina Resolver.](#)

In AWS AppSync, il resolver è legato a un runtime, che è l'ambiente in cui viene eseguito. I runtime determinano la lingua in cui verrà scritto il resolver. Attualmente sono supportati due runtime: APPSYNC_JS (JavaScript) e Apache Velocity Template Language (VTL).

Quando si implementano i resolver, esiste una struttura generale che seguono:

- Prima della fase: quando viene effettuata una richiesta dal client, ai resolver per i campi dello schema utilizzati (in genere le query, le mutazioni, le sottoscrizioni) vengono trasmessi i dati della richiesta. Il resolver inizierà a elaborare i dati della richiesta con un gestore del passaggio precedente, che consente di eseguire alcune operazioni di preelaborazione prima che i dati vengano trasferiti nel resolver.
- Funzione/i: dopo l'esecuzione del passaggio precedente, la richiesta viene passata all'elenco delle funzioni. La prima funzione dell'elenco verrà eseguita sulla fonte di dati. Una funzione è un sottoinsieme del codice del resolver contenente il proprio gestore di richieste e risposte. Un gestore di richieste prenderà i dati della richiesta ed eseguirà operazioni sulla fonte dei dati. Il gestore della risposta elaborerà la risposta dell'origine dati prima di restituirla all'elenco. Se è presente più di una funzione, i dati della richiesta verranno inviati alla funzione successiva nell'elenco da eseguire. Le funzioni nell'elenco verranno eseguite in serie nell'ordine definito dallo sviluppatore. Una volta eseguite tutte le funzioni, il risultato finale viene passato alla fase successiva.

- Dopo la fase: la fase successiva è una funzione di gestione che consente di eseguire alcune operazioni finali sulla risposta finale della funzione prima di passarla alla risposta GraphQL.

Questo flusso è un esempio di risolutore di pipeline. I resolver di pipeline sono supportati in entrambi i runtime. Tuttavia, questa è una spiegazione semplificata di cosa possono fare i resolver di pipeline. Inoltre, stiamo descrivendo solo una possibile configurazione del resolver. [Per ulteriori informazioni sulle configurazioni dei resolver supportate, consulta la panoramica dei resolver per JavaScript APPSYNC_JS o la panoramica dei modelli di mappatura Resolver per VTL.](#)

Come puoi vedere, i resolver sono modulari. Affinché i componenti del resolver funzionino correttamente, devono essere in grado di esaminare lo stato di esecuzione da altri componenti. Dalla sezione [Resolver](#), sapete che a ogni componente del resolver possono essere trasmesse informazioni vitali sullo stato dell'esecuzione sotto forma di un insieme di argomenti (,, ecc.). `args` `context` Nel AWS AppSync, questo è gestito rigorosamente da. `context` È un contenitore per le informazioni sul campo da risolvere. Ciò può includere qualsiasi cosa, dagli argomenti passati, ai risultati, ai dati di autorizzazione, ai dati di intestazione, ecc. Per ulteriori informazioni sul contesto, vedere il riferimento al contesto [Resolver Context Object per APPSYNC_JS](#) o il riferimento contestuale del modello di mappatura [Resolver](#) per VTL.

Il contesto non è l'unico strumento che puoi usare per implementare il tuo resolver. AWS AppSync supporta un'ampia gamma di utilità per la generazione di valore, la gestione degli errori, l'analisi, la conversione, ecc. [Puoi vedere un elenco di utilità qui per APPSYNC_JS o qui per VTL.](#)

Nelle sezioni seguenti, imparerai come configurare i resolver nella tua API GraphQL.

Argomenti

- [Creazione di interrogazioni di base \(\) JavaScript](#)
- [Creazione di interrogazioni di base \(VTL\)](#)

Creazione di interrogazioni di base () JavaScript

I resolver GraphQL connettono i campi nello schema di un tipo a un'origine dati. I resolver sono il meccanismo mediante il quale vengono soddisfatte le richieste.

Resolver AWS AppSync utilizzati JavaScript per convertire un'espressione GraphQL in un formato utilizzabile dall'origine dati. In alternativa, i modelli di mappatura possono essere scritti in [Apache Velocity Template Language \(VTL\)](#) per convertire un'espressione GraphQL in un formato utilizzabile dall'origine dati.

Questa sezione descrive come configurare i resolver utilizzando JavaScript. La sezione [Resolver tutorials \(JavaScript\) fornisce tutorial](#) approfonditi su come implementare i resolver utilizzando JavaScript. La sezione [Resolver reference \(JavaScript\) fornisce una spiegazione delle operazioni di utilità che possono essere utilizzate con i resolver](#). JavaScript

Consigliamo di seguire questa guida prima di tentare di utilizzare uno qualsiasi dei tutorial sopra citati.

In questa sezione, spiegheremo come creare e configurare resolver per query e mutazioni.

Note

Questa guida presuppone che tu abbia creato il tuo schema e che tu abbia almeno una query o una mutazione. [Se stai cercando abbonamenti \(dati in tempo reale\), consulta questa guida.](#)

In questa sezione, forniremo alcuni passaggi generali per la configurazione dei resolver insieme a un esempio che utilizza lo schema seguente:

```
// schema.graphql file

input CreatePostInput {
  title: String!
  date: AWSDateTime!
}

type Post {
  id: ID!
  title: String!
  date: AWSDateTime!
}

type Mutation {
  createPost(input: CreatePostInput!): Post!
}

type Query {
  getPost(id: ID!): Post
}
```

Creazione di risolutori di query di base

Questa sezione ti mostrerà come creare un risolutore di query di base.

Console

1. [Accedi a Console di gestione AWS e apri la AppSync console.](#)
 - a. Nella APIs dashboard, scegli la tua API GraphQL.
 - b. Nella barra laterale, scegli Schema.
2. Inserisci i dettagli dello schema e della fonte di dati. Per ulteriori informazioni, consulta [le sezioni Progettazione dello schema](#) e [Collegamento di un'origine dati](#).
3. Accanto all'editor dello schema, c'è una finestra chiamata Resolvers. Questa casella contiene un elenco dei tipi e dei campi definiti nella finestra Schema. È possibile allegare resolver ai campi. Molto probabilmente allegherai dei resolver alle tue operazioni sul campo. In questa sezione, esamineremo semplici configurazioni di query. Nella sezione Tipo di query, scegli Allega accanto al campo della query.
4. Nella pagina Attach resolver, in Tipo di resolver, puoi scegliere tra resolver pipeline o unit. [Per ulteriori informazioni su questi tipi, consulta Resolvers](#). Questa guida utilizzerà. pipeline resolvers

Tip

Durante la creazione di risolutori di pipeline, le sorgenti dati verranno allegate alle funzioni della pipeline. Le funzioni vengono create dopo aver creato il risolutore della pipeline stesso, motivo per cui non è possibile impostarlo in questa pagina. Se utilizzi un risolutore di unità, la fonte di dati è collegata direttamente al resolver, quindi devi impostarla in questa pagina.

Per Resolver runtime, scegli di abilitare il runtime. APPSYNC_JS JavaScript

5. Puoi abilitare la [memorizzazione nella cache per questa API](#). Ti consigliamo di disattivare questa funzionalità per ora. Scegli Create (Crea).
6. Nella pagina Modifica resolver, c'è un editor di codice chiamato Resolver code che consente di implementare la logica per il gestore e la risposta del resolver (prima e dopo i passaggi). [Per ulteriori informazioni, consulta la panoramica dei resolver. JavaScript](#)

Note

[Nel nostro esempio, lasceremo la richiesta vuota e la risposta impostata per restituire l'ultima fonte di dati risultante dal contesto:](#)

```
import {util} from '@aws-appsync/utils';

export function request(ctx) {
  return {};
}

export function response(ctx) {
  return ctx.prev.result;
}
```

Sotto questa sezione, c'è una tabella chiamata Funzioni. Le funzioni consentono di implementare codice che può essere riutilizzato su più resolver. Invece di riscrivere o copiare costantemente il codice, puoi memorizzare il codice sorgente come funzione da aggiungere a un resolver ogni volta che ne hai bisogno.

Le funzioni costituiscono la maggior parte dell'elenco delle operazioni di una pipeline. Quando si utilizzano più funzioni in un resolver, si imposta l'ordine delle funzioni e queste verranno eseguite in quell'ordine in sequenza. Vengono eseguiti dopo l'esecuzione della funzione di richiesta e prima dell'inizio della funzione di risposta.

Per aggiungere una nuova funzione, in Funzioni, scegli Aggiungi funzione, quindi Crea nuova funzione. In alternativa, potresti invece visualizzare il pulsante Crea funzione tra cui scegliere.

- a. Scegli una fonte di dati. Questa sarà la fonte di dati su cui agisce il resolver.

 Note

Nel nostro esempio, stiamo collegando un resolver `getPost`, che recupera un oggetto tramite `Post id`. Supponiamo di aver già impostato una tabella DynamoDB per questo schema. La sua chiave di partizione è impostata su `id` ed è vuota.

- b. Inserisci `unFunction name`.
- c. In Codice funzione, dovrai implementare il comportamento della funzione. Ciò potrebbe creare confusione, ma ogni funzione avrà il proprio gestore locale di richieste e risposte. La richiesta viene eseguita, quindi viene effettuata la chiamata all'origine dati per gestire la richiesta, quindi la risposta dell'origine dati viene elaborata dal gestore della risposta.

[Il risultato viene archiviato nell'oggetto contestuale](#). Successivamente, verrà eseguita la funzione successiva nell'elenco o verrà passata al gestore di risposte post-passaggio se è l'ultima.

Note

Nel nostro esempio, stiamo collegando un resolver `agetPost`, che ottiene un elenco di `Post` oggetti dalla fonte di dati. La nostra funzione di richiesta richiederà i dati dalla nostra tabella, la tabella passerà la sua risposta al contesto (`ctx`), quindi la risposta restituirà il risultato nel contesto. AWS AppSync la forza di questa soluzione risiede nella sua interconnessione con altri servizi. AWS Poiché utilizziamo DynamoDB, disponiamo di [una suite di](#) operazioni per semplificare cose come queste. Abbiamo anche alcuni esempi standard per altri tipi di fonti di dati.

Il nostro codice sarà simile a questo:

```
import { util } from '@aws-appsync/utils';

/**
 * Performs a scan on the dynamodb data source
 */
export function request(ctx) {
  return { operation: 'Scan' };
}

/**
 * return a list of scanned post items
 */
export function response(ctx) {
  return ctx.result.items;
}
```

In questo passaggio, abbiamo aggiunto due funzioni:

- `request`: Il gestore della richiesta esegue l'operazione di recupero sulla fonte di dati. L'argomento contiene l'oggetto context (`ctx`) o alcuni dati disponibili per tutti i resolver che eseguono una particolare operazione. Ad esempio, potrebbe contenere dati di autorizzazione, i nomi dei campi da risolvere, ecc. L'istruzione `return` esegue un'[Scan](#)operazione (vedi [qui](#) per alcuni esempi). Poiché lavoriamo con DynamoDB, siamo autorizzati a utilizzare alcune delle

operazioni di quel servizio. La scansione esegue un recupero di base di tutti gli elementi della nostra tabella. Il risultato di questa operazione viene archiviato nell'oggetto di contesto come `result` contenitore prima di essere passato al gestore delle risposte. `request` Viene eseguito prima della risposta nella pipeline.

- `response`: Il gestore di risposte che restituisce l'output di `request`. L'argomento è l'oggetto di contesto aggiornato e l'istruzione `return ctx.prev.result`. A questo punto della guida, potresti non avere familiarità con questo valore. `ctx` si riferisce all'oggetto contestuale. `prev` si riferisce all'operazione precedente nella pipeline, che era la nostra `request`. `result` Contiene i risultati del resolver mentre si muove attraverso la pipeline. Se metti tutto insieme, `ctx.prev.result` restituisce il risultato dell'ultima operazione eseguita, che era il gestore della richiesta.

d. Scegli Crea dopo aver finito.

7. Tornando alla schermata del resolver, in Funzioni, scegli il menu a discesa Aggiungi funzione e aggiungi la tua funzione all'elenco delle funzioni.
8. Scegli Salva per aggiornare il resolver.

CLI

Per aggiungere la tua funzione

- Crea una funzione per il tuo risolutore di pipeline usando il comando. [create-function](#)

Dovrai inserire alcuni parametri per questo particolare comando:

1. La `api-id` della tua API.
2. `name` La funzione nella AWS AppSync console.
3. Il `data-source-name`, o il nome dell'origine dati che verrà utilizzata dalla funzione. Deve essere già stato creato e collegato all'API GraphQL nel AWS AppSync servizio.
4. L'`runtimeambiente` o il linguaggio della funzione. Perché JavaScript, il nome deve essere `APPSYNC_JS`, e il runtime, `1.0.0`.
5. I code gestori di richieste e risposte della funzione. Sebbene sia possibile digitarlo manualmente, è molto più semplice aggiungerlo a un file.txt (o un formato simile) e quindi passarlo come argomento.

Note

Il nostro codice di query sarà contenuto in un file passato come argomento:

```
import { util } from '@aws-appsync/utils';

/**
 * Performs a scan on the dynamodb data source
 */
export function request(ctx) {
  return { operation: 'Scan' };
}

/**
 * return a list of scanned post items
 */
export function response(ctx) {
  return ctx.result.items;
}
```

Un comando di esempio può essere simile al seguente:

```
aws appsync create-function \
--api-id abcdefghijklmnopqrstuvwxyz \
--name get_posts_func_1 \
--data-source-name table-for-posts \
--runtime name=APPSYNC_JS,runtimeVersion=1.0.0 \
--code file://~/path/to/file/{filename}.{fileType}
```

Verrà restituito un output nella CLI. Ecco un esempio:

```
{
  "functionConfiguration": {
    "functionId": "ejglgvmcabdn7lx75ref4qeig4",
    "functionArn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz/functions/ejglgvmcabdn7lx75ref4qeig4",
    "name": "get_posts_func_1",
    "dataSourceName": "table-for-posts",
    "maxBatchSize": 0,
```

```
    "runtime": {  
      "name": "APPSYNC_JS",  
      "runtimeVersion": "1.0.0"  
    },  
    "code": "Code output goes here"  
  }  
}
```

Note

Assicurati di registrarlo `functionId` da qualche parte poiché verrà utilizzato per collegare la funzione al resolver.

Per creare il tuo resolver

- Crea una funzione di pipeline Query eseguendo il comando. [create-resolver](#)

Dovrai inserire alcuni parametri per questo particolare comando:

1. La `api-id` della tua API.
2. Il `type-name`, o il tipo di oggetto speciale nel tuo schema (Query, Mutation, Subscription).
3. Il `field-name`, o l'operazione di campo all'interno del tipo di oggetto speciale a cui si desidera collegare il resolver.
4. Il `kind`, che specifica un'unità o un resolver di pipeline. Impostalo per abilitare le funzioni della PIPELINE pipeline.
5. La `pipeline-config`, o la o le funzioni da collegare al resolver. Assicurati di conoscere `functionId` i valori delle tue funzioni. L'ordine delle inserzioni è importante.
6. Il `runtime`, che era APPSYNC_JS (JavaScript). `runtimeVersion` Attualmente è 1.0.0.
7. Il `code`, che contiene i gestori della fase prima e dopo.

Note

Il nostro codice di query sarà in un file passato come argomento:

```
import { util } from '@aws-appsync/utils';  
  
/**
```

```

* Sends a request to `put` an item in the DynamoDB data source
*/
export function request(ctx) {
  const { id, ...values } = ctx.args;
  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({ id }),
    attributeValues: util.dynamodb.toMapValues(values),
  };
}

/**
 * returns the result of the `put` operation
 */
export function response(ctx) {
  return ctx.result;
}

```

Un comando di esempio può essere simile al seguente:

```

aws appsync create-resolver \
--api-id abcdefghijklmnopqrstuvwxyz \
--type-name Query \
--field-name getPost \
--kind PIPELINE \
--pipeline-config functions=ejglgvmcabdn7lx75ref4qeig4 \
--runtime name=APPSYNC_JS,runtimeVersion=1.0.0 \
--code file:///path/to/file/{filename}.{fileType}

```

Verrà restituito un output nella CLI. Ecco un esempio:

```

{
  "resolver": {
    "typeName": "Mutation",
    "fieldName": "getPost",
    "resolverArn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz/types/Mutation/resolvers/getPost",
    "kind": "PIPELINE",
    "pipelineConfig": {
      "functions": [
        "ejglgvmcabdn7lx75ref4qeig4"
      ]
    }
  }
}

```

```

    ]
  },
  "maxBatchSize": 0,
  "runtime": {
    "name": "APPSYNC_JS",
    "runtimeVersion": "1.0.0"
  },
  "code": "Code output goes here"
}
}

```

CDK

Tip

[Prima di utilizzare il CDK, consigliamo di consultare la documentazione ufficiale del CDK insieme AWS AppSync al riferimento CDK.](#)

I passaggi elencati di seguito mostreranno solo un esempio generale dello snippet utilizzato per aggiungere una particolare risorsa. Questa non è pensata per essere una soluzione funzionante nel codice di produzione. Supponiamo inoltre che tu abbia già un'app funzionante.

Un'app di base avrà bisogno delle seguenti cose:

1. Direttive di importazione dei servizi
2. Codice dello schema
3. Generatore di fonti di dati
4. Codice di funzione
5. Codice del resolver

Dalle sezioni [Progettazione dello schema](#) e [Allegare una fonte di dati](#), sappiamo che il file stack includerà le direttive di importazione del modulo:

```

import * as x from 'x'; # import wildcard as the 'x' keyword from 'x-service'
import {a, b, ...} from 'c'; # import {specific constructs} from 'c-service'

```

Note

Nelle sezioni precedenti, abbiamo solo indicato come importare i costrutti. AWS AppSync In codice reale, dovrai importare più servizi solo per eseguire l'app. Nel nostro esempio, se dovessimo creare un'app CDK molto semplice, importeremmo almeno il AWS AppSync servizio insieme alla nostra fonte di dati, che era una tabella DynamoDB. Dovremmo anche importare alcuni costrutti aggiuntivi per distribuire l'app:

```
import * as cdk from 'aws-cdk-lib';
import * as appsync from 'aws-cdk-lib/aws-appsync';
import * as dynamodb from 'aws-cdk-lib/aws-dynamodb';
import { Construct } from 'constructs';
```

Per riassumere ognuno di questi:

- `import * as cdk from 'aws-cdk-lib';`: Ciò consente di definire l'app CDK e i costrutti come lo stack. Contiene anche alcune utili funzioni di utilità per la nostra applicazione, come la manipolazione dei metadati. [Se conosci questa direttiva di importazione, ma ti stai chiedendo perché la libreria di base cdk non viene utilizzata qui, consulta la pagina Migrazione.](#)
- `import * as appsync from 'aws-cdk-lib/aws-appsync';`: Questo importa il [AWS AppSync servizio](#).
- `import * as dynamodb from 'aws-cdk-lib/aws-dynamodb';`: Importa il servizio [DynamoDB](#).
- `import { Construct } from 'constructs';`: [Ne abbiamo bisogno per definire il costrutto root.](#)

Il tipo di importazione dipende dai servizi che stai chiamando. Ti consigliamo di consultare la documentazione CDK per alcuni esempi. Lo schema nella parte superiore della pagina sarà un file separato nell'app CDK come `.graphql` file. Nel file stack, possiamo associarlo a un nuovo GraphQL utilizzando il modulo:

```
const add_api = new appsync.GraphqlApi(this, 'graphql-example', {
  name: 'my-first-api',
  schema: appsync.SchemaFile.fromAsset(path.join(__dirname, 'schema.graphql')),
});
```

Note

Nell'ambito `add_api`, stiamo aggiungendo una nuova API GraphQL utilizzando la new parola chiave seguita da `appsync.GraphqlApi(scope: Construct, id: string, props: GraphqlApiProps)`. Il nostro ambito è `this`, l'id CFN è `graphql-example` e i nostri oggetti di scena sono `my-first-api` (nome dell'API nella console) e `schema.graphql` (il percorso assoluto del file di schema).

Per aggiungere una fonte di dati, devi prima aggiungere la tua fonte di dati allo stack. Quindi, è necessario associarlo all'API GraphQL utilizzando il metodo specifico del codice sorgente. L'associazione avverrà quando farai funzionare il resolver. Nel frattempo, facciamo un esempio creando la tabella DynamoDB usando: `dynamodb.Table`

```
const add_ddb_table = new dynamodb.Table(this, 'posts-table', {
  partitionKey: {
    name: 'id',
    type: dynamodb.AttributeType.STRING,
  },
});
```

Note

Se dovessimo utilizzarlo nel nostro esempio, aggiungeremmo una nuova tabella DynamoDB con l'id CFN `posts-table` di e una chiave di partizione di `id` (S)

Successivamente, dobbiamo implementare il nostro resolver nel file `stack`. Ecco un esempio di una semplice query che analizza tutti gli elementi in una tabella DynamoDB:

```
const add_func = new appsync.AppsyncFunction(this, 'func-get-posts', {
  name: 'get_posts_func_1',
  add_api,
  dataSource: add_api.addDynamoDbDataSource('table-for-posts', add_ddb_table),
  code: appsync.Code.fromInline(`
    export function request(ctx) {
      return { operation: 'Scan' };
    }

    export function response(ctx) {
```



```

        return ctx.result.items;
    }
    `),
    runtime: appsync.FunctionRuntime.JS_1_0_0,
  });

new appsync.Resolver(this, 'pipeline-resolver-get-posts', {
  add_api,
  typeName: 'Query',
  fieldName: 'getPost',
  code: appsync.Code.fromInline(`
    export function request(ctx) {
      return {};
    }

    export function response(ctx) {
      return ctx.prev.result;
    }
  `),
  runtime: appsync.FunctionRuntime.JS_1_0_0,
  pipelineConfig: [add_func],
});

```

Note

Innanzitutto, abbiamo creato una funzione chiamata. `add_func` Questo ordine di creazione può sembrare un po' controintuitivo, ma è necessario creare le funzioni nel resolver della pipeline prima di creare il resolver stesso. Una funzione segue la forma:

```
AppsyncFunction(scope: Construct, id: string, props: AppsyncFunctionProps)
```

Il nostro ambito era `this`, il nostro ID CFN lo era `func-get-posts` e i nostri oggetti di scena contenevano i dettagli effettivi della funzione. All'interno degli oggetti di scena, abbiamo incluso:

- La name funzione che sarà presente nella AWS AppSync console (`get_posts_func_1`).
- L'API GraphQL che abbiamo creato in precedenza (`add_api`).
- L'origine dei dati; questo è il punto in cui colleghiamo l'origine dati al valore dell'API GraphQL, quindi la colleghiamo alla funzione. Prendiamo la tabella che abbiamo creato

(`add_ddb_table`) e la colleghiamo all'API GraphQL (`add_api`) utilizzando uno dei `GraphQLApi` metodi ([addDynamoDbDataSource](#)). Il valore `id` (`table-for-posts`) è il nome dell'origine dati nella AWS AppSync console. Per un elenco di metodi specifici dell'origine, consulta le pagine seguenti:

- [DynamoDbDataSource](#)
 - [EventBridgeDataSource](#)
 - [HttpDataSource](#)
 - [LambdaDataSource](#)
 - [NoneDataSource](#)
 - [OpenSearchDataSource](#)
 - [RdsDataSource](#)
- Il codice contiene i gestori di richiesta e risposta della nostra funzione, che consistono in una semplice scansione e restituzione.
 - Il runtime specifica che vogliamo usare la versione di runtime `APPSYNC_JS 1.0.0`. Nota che questa è attualmente l'unica versione disponibile per `APPSYNC_JS`.

Successivamente, dobbiamo collegare la funzione al risolutore della pipeline. Abbiamo creato il nostro resolver utilizzando il modulo:

```
Resolver(scope: Construct, id: string, props: ResolverProps)
```

Il nostro ambito era `this`, il nostro ID CFN lo era `pipeline-resolver-get-posts` e i nostri oggetti di scena contenevano i dettagli effettivi della funzione. All'interno degli oggetti di scena, abbiamo incluso:

- L'API GraphQL che abbiamo creato in precedenza (`add_api`).
- Il nome speciale del tipo di oggetto; si tratta di un'operazione di interrogazione, quindi abbiamo semplicemente aggiunto il valore `Query`.
- Il nome del campo (`getPost`) è il nome del campo nello schema sotto il `Query` tipo.
- Il codice contiene i gestori prima e dopo. Il nostro esempio restituisce semplicemente i risultati presenti nel contesto dopo che la funzione ha eseguito la sua operazione.
- Il runtime specifica che vogliamo usare la versione di runtime `APPSYNC_JS 1.0.0`. Nota che questa è attualmente l'unica versione disponibile per `APPSYNC_JS`.

- La configurazione della pipeline contiene il riferimento alla funzione che abbiamo creato `()`. `add_func`

Per riassumere ciò che è successo in questo esempio, hai visto una AWS AppSync funzione che implementava un gestore di richieste e risposte. La funzione era responsabile dell'interazione con la fonte dei dati. Il gestore della richiesta ha inviato un'operazione a AWS AppSync, indicandogli quale operazione eseguire sull'origine dati DynamoDB. Il gestore della risposta ha restituito l'elenco degli elementi `()`. `ctx.result.items` L'elenco degli elementi è stato quindi mappato automaticamente al tipo `Post` GraphQL.

Creazione di risolutori di mutazioni di base

Questa sezione ti mostrerà come creare un risolutore di mutazioni di base.

Console

1. [Accedi a Console di gestione AWS e apri la console. AppSync](#)
 - a. Nella APIs dashboard, scegli la tua API GraphQL.
 - b. Nella barra laterale, scegli Schema.
2. Nella sezione Resolver e nel tipo di mutazione, scegli `Allega` accanto al tuo campo.

Note

Nel nostro esempio, stiamo collegando un resolver `createPost`, che aggiunge un oggetto alla nostra tabella. `Post` Supponiamo di utilizzare la stessa tabella DynamoDB dell'ultima sezione. La sua chiave di partizione è impostata su `id` ed è vuota.

3. Nella pagina `Allega resolver`, in `Tipo di risolutore`, scegli. `pipeline resolvers` [Come promemoria, puoi trovare ulteriori informazioni sui resolver qui](#). Per Resolver runtime, scegli di abilitare il runtime. `APPSYNC_JS JavaScript`
4. Puoi abilitare la [memorizzazione nella cache per questa API](#). Ti consigliamo di disattivare questa funzionalità per ora. Scegli `Create (Crea)`.
5. Scegli `Aggiungi funzione`, quindi scegli `Crea nuova funzione`. In alternativa, potresti invece visualizzare un pulsante `Crea funzione` tra cui scegliere.

- a. Scegli l'origine dati. Questa dovrebbe essere la fonte di cui manipolerai i dati con la mutazione.
- b. Inserisci un. `Function name`
- c. In Codice funzione, dovrai implementare il comportamento della funzione. Questa è una mutazione, quindi la richiesta eseguirà idealmente alcune operazioni di modifica dello stato sulla fonte di dati richiamata. Il risultato verrà elaborato dalla funzione di risposta.

Note

`createPost` sta aggiungendo, o «inserendo», un nuovo Post nella tabella con i nostri parametri come dati. Potremmo aggiungere qualcosa del genere:

```
import { util } from '@aws-appsync/utils';

/**
 * Sends a request to `put` an item in the DynamoDB data source
 */
export function request(ctx) {
  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({id: util.autoId()}),
    attributeValues: util.dynamodb.toMapValues(ctx.args.input),
  };
}

/**
 * returns the result of the `put` operation
 */
export function response(ctx) {
  return ctx.result;
}
```

In questo passaggio, abbiamo anche aggiunto request response funzioni:

- `request`: Il gestore della richiesta accetta il contesto come argomento. L'istruzione `return` del gestore di richieste esegue un [PutItem](#) comando, che è un'operazione DynamoDB integrata ([vedi](#) [qui](#) [o](#) [qui](#) per alcuni esempi). Il `PutItem` comando aggiunge un Post oggetto alla nostra tabella DynamoDB prendendo il valore della key partizione (generato automaticamente

`util.autoid()` da `attributes` e dall'input dell'argomento di contesto (questi sono i valori che passeremo nella nostra richiesta). Gli argomenti `key` è `id` e `attributes` sono gli `date` e `title` i campi. Sono entrambi preformattati tramite l'[util.dynamodb.toMapValues](#) helper per funzionare con la tabella DynamoDB.

- `response`: La risposta accetta il contesto aggiornato e restituisce il risultato del gestore della richiesta.

d. Scegli Crea dopo aver finito.

6. Tornando alla schermata del resolver, in Funzioni, scegli il menu a discesa Aggiungi funzione e aggiungi la tua funzione all'elenco delle funzioni.
7. Scegli Salva per aggiornare il resolver.

CLI

Per aggiungere la tua funzione

- Crea una funzione per il tuo risolutore di pipeline usando il comando. [create-function](#)

Dovrai inserire alcuni parametri per questo particolare comando:

1. La `api-id` della tua API.
2. `name` La funzione nella AWS AppSync console.
3. Il `data-source-name`, o il nome dell'origine dati che verrà utilizzata dalla funzione. Deve essere già stato creato e collegato all'API GraphQL nel AWS AppSync servizio.
4. L'`runtimeambiente` o il linguaggio della funzione. Perché JavaScript, il nome deve essere `APPSYNC_JS`, e il runtime, `1.0.0`.
5. I code gestori di richieste e risposte della funzione. Sebbene sia possibile digitarlo manualmente, è molto più semplice aggiungerlo a un file.txt (o un formato simile) e passarlo come argomento.

Note

Il nostro codice di query sarà contenuto in un file passato come argomento:

```
import { util } from '@aws-appsync/utils';
```

```

/**
 * Sends a request to `put` an item in the DynamoDB data source
 */
export function request(ctx) {
  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({id: util.autoId()}),
    attributeValues: util.dynamodb.toMapValues(ctx.args.input),
  };
}

/**
 * returns the result of the `put` operation
 */
export function response(ctx) {
  return ctx.result;
}

```

Un comando di esempio può essere simile al seguente:

```

aws appsync create-function \
--api-id abcdefghijklmnopqrstuvwxyz \
--name add_posts_func_1 \
--data-source-name table-for-posts \
--runtime name=APPSYNC_JS,runtimeVersion=1.0.0 \
--code file:///path/to/file/{filename}.{fileType}

```

Verrà restituito un output nella CLI. Ecco un esempio:

```

{
  "functionConfiguration": {
    "functionId": "vulcmbfcxffiram63psb4ddua",
    "functionArn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz/functions/vulcmbfcxffiram63psb4ddua",
    "name": "add_posts_func_1",
    "dataSourceName": "table-for-posts",
    "maxBatchSize": 0,
    "runtime": {
      "name": "APPSYNC_JS",
      "runtimeVersion": "1.0.0"
    },
  },
}

```

```
    "code": "Code output goes here"  
  }  
}
```

Note

Assicurati di registrarlo `functionId` da qualche parte poiché verrà utilizzato per collegare la funzione al resolver.

Per creare il tuo resolver

- Crea una funzione di pipeline Mutation eseguendo il comando. [create-resolver](#)

Dovrai inserire alcuni parametri per questo particolare comando:

1. La `api-id` della tua API.
2. Il `type-name`, o il tipo di oggetto speciale nel tuo schema (Query, Mutation, Subscription).
3. Il `field-name`, o l'operazione di campo all'interno del tipo di oggetto speciale a cui si desidera collegare il resolver.
4. Il `kind`, che specifica un'unità o un resolver di pipeline. Impostalo per abilitare le funzioni della PIPELINE pipeline.
5. La `pipeline-config`, o la o le funzioni da collegare al resolver. Assicurati di conoscere `functionId` i valori delle tue funzioni. L'ordine delle inserzioni è importante.
6. Il `runtime`, che era APPSYNC_JS (JavaScript). `runtimeVersion` Attualmente è 1.0.0.
7. Il `code`, che contiene il passaggio prima e dopo.

Note

Il nostro codice di query sarà in un file passato come argomento:

```
import { util } from '@aws-appsync/utils';  
  
/**  
 * Sends a request to `put` an item in the DynamoDB data source  
 */  
export function request(ctx) {  
  const { id, ...values } = ctx.args;
```

```

    return {
      operation: 'PutItem',
      key: util.dynamodb.toMapValues({ id }),
      attributeValues: util.dynamodb.toMapValues(values),
    };
  }

  /**
   * returns the result of the `put` operation
   */
  export function response(ctx) {
    return ctx.result;
  }

```

Un comando di esempio può essere simile al seguente:

```

aws appsync create-resolver \
--api-id abcdefghijklmnopqrstuvwxyz \
--type-name Mutation \
--field-name createPost \
--kind PIPELINE \
--pipeline-config functions=vulcmbfcxffiram63psb4ddua \
--runtime name=APPSYNC_JS,runtimeVersion=1.0.0 \
--code file:///path/to/file/{filename}.{fileType}

```

Verrà restituito un output nella CLI. Ecco un esempio:

```

{
  "resolver": {
    "typeName": "Mutation",
    "fieldName": "createPost",
    "resolverArn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz/types/Mutation/resolvers/createPost",
    "kind": "PIPELINE",
    "pipelineConfig": {
      "functions": [
        "vulcmbfcxffiram63psb4ddua"
      ]
    },
    "maxBatchSize": 0,
    "runtime": {

```



```

      "name": "APPSYNC_JS",
      "runtimeVersion": "1.0.0"
    },
    "code": "Code output goes here"
  }
}

```

CDK

Tip

Prima di utilizzare il CDK, consigliamo di consultare la documentazione ufficiale del CDK insieme AWS AppSync al riferimento CDK.

I passaggi elencati di seguito mostreranno solo un esempio generale dello snippet utilizzato per aggiungere una particolare risorsa. Questa non è pensata per essere una soluzione funzionante nel codice di produzione. Supponiamo inoltre che tu abbia già un'app funzionante.

- Per effettuare una mutazione, supponendo che siate nello stesso progetto, potete aggiungerla allo stack file come nella query. Ecco una funzione e un resolver modificati per una mutazione che ne aggiunge una nuova alla tabella: Post

```

const add_func_2 = new appsync.AppsyncFunction(this, 'func-add-post', {
  name: 'add_posts_func_1',
  add_api,
  dataSource: add_api.addDynamoDbDataSource('table-for-posts-2', add_ddb_table),
  code: appsync.Code.fromInline(`
    export function request(ctx) {
      return {
        operation: 'PutItem',
        key: util.dynamodb.toMapValues({id: util.autoId()}),
        attributeValues: util.dynamodb.toMapValues(ctx.args.input),
      };
    }

    export function response(ctx) {
      return ctx.result;
    }
  `),

```

```

    runtime: appsync.FunctionRuntime.JS_1_0_0,
  });

new appsync.Resolver(this, 'pipeline-resolver-create-posts', {
  add_api,
  typeName: 'Mutation',
  fieldName: 'createPost',
  code: appsync.Code.fromInline(`
    export function request(ctx) {
      return {};
    }

    export function response(ctx) {
      return ctx.prev.result;
    }
  `),
  runtime: appsync.FunctionRuntime.JS_1_0_0,
  pipelineConfig: [add_func_2],
});

```

Note

Poiché questa mutazione e la query sono strutturate in modo simile, ci limiteremo a spiegare le modifiche che abbiamo apportato per apportare la mutazione.

Nella funzione, abbiamo cambiato l'id CFN in `func-add-post` e il nome in `add_posts_func_1` per riflettere il fatto che stiamo aggiungendo Posts alla tabella. Nell'origine dati, abbiamo creato una nuova associazione alla nostra table (`add_ddb_table`) nella AWS AppSync console `table-for-posts-2` perché il `addDynamoDbDataSource` metodo lo richiede. Tieni presente che questa nuova associazione utilizza ancora la stessa tabella che abbiamo creato in precedenza, ma ora abbiamo due connessioni ad essa nella AWS AppSync console: una per la query `as table-for-posts` e una per la mutazione `astable-for-posts-2`. Il codice è stato modificato per aggiungere un valore `Post` generando automaticamente il suo `id` valore e accettando l'input di un client per il resto dei campi.

Nel resolver, abbiamo modificato il valore `id` in modo `pipeline-resolver-create-posts` che rifletta il fatto che stiamo aggiungendo Posts alla tabella. Per riflettere la mutazione nello schema, il nome del tipo è stato cambiato in `Mutation`, e il nome, `createPost`. La configurazione della pipeline è stata impostata sulla nostra nuova funzione di mutazione. `add_func_2`

Per riassumere ciò che sta accadendo in questo esempio, converte AWS AppSync automaticamente gli argomenti definiti nel `createPost` campo dallo schema GraphQL in operazioni DynamoDB. L'esempio archivia i record in DynamoDB utilizzando una chiave `id` di, che viene creata automaticamente utilizzando il nostro helper. `util.autoId()` Tutti gli altri campi passati agli argomenti di contesto (`ctx.args.input`) dalle richieste effettuate nella AWS AppSync console o in altro modo verranno memorizzati come attributi della tabella. Sia la chiave che gli attributi vengono mappati automaticamente su un formato DynamoDB compatibile utilizzando l'helper. `util.dynamodb.toMapValues(values)`

AWS AppSync supporta anche flussi di lavoro di test e debug per la modifica dei resolver. È possibile utilizzare un `context` oggetto fittizio per vedere il valore trasformato del modello prima di richiamarlo. Facoltativamente, puoi visualizzare la richiesta completa a un'origine dati in modo interattivo quando esegui una query. [Per ulteriori informazioni, vedete Test and debug resolvers \(\) JavaScript e Monitoring and logging.](#)

Resolver avanzati

Se stai seguendo la sezione opzionale sull'impaginazione in [Progettazione dello schema](#), devi comunque aggiungere il resolver alla richiesta per utilizzare l'impaginazione. Il nostro esempio ha utilizzato una paginazione di query chiamata `getPosts` per restituire solo una parte degli elementi richiesti alla volta. Il codice del nostro resolver su quel campo potrebbe essere simile al seguente:

```
/**
 * Performs a scan on the dynamodb data source
 */
export function request(ctx) {
  const { limit = 20, nextToken } = ctx.args;
  return { operation: 'Scan', limit, nextToken };
}

/**
 * @returns the result of the `put` operation
 */
export function response(ctx) {
  const { items: posts = [], nextToken } = ctx.result;
  return { posts, nextToken };
}
```

Nella richiesta, passiamo nel contesto della richiesta. Il nostro `limit` è **20**, il che significa che restituiamo fino a 20 `Posts` nella prima query. Il nostro `nextToken` cursore è fissato alla prima `Post`

voce nella fonte di dati. Questi vengono passati agli args. La richiesta esegue quindi una scansione dalla prima Post fino al numero limite di scansione. L'origine dati memorizza il risultato nel contesto, che viene passato alla risposta. La risposta Posts restituisce la risposta recuperata, quindi imposta l'impostazione nextToken è impostata sulla Post voce subito dopo il limite. La richiesta successiva viene inviata per fare esattamente la stessa cosa, ma a partire dall'offset subito dopo la prima query. Tieni presente che questo tipo di richieste vengono eseguite in sequenza e non in parallelo.

Test e debug dei resolver in () AWS AppSync JavaScript

AWS AppSync esegue resolver su un campo GraphQL su un'origine dati. Quando si lavora con i resolver per pipeline, le funzioni interagiscono con le fonti di dati. Come descritto nella [panoramica dei JavaScript resolver, le funzioni comunicano con le fonti di dati utilizzando gestori](#) di richieste e risposte scritti ed eseguiti nel runtime. JavaScript APPSYNC_JS Ciò consente di fornire logica e condizioni personalizzate prima e dopo la comunicazione con l'origine dati.

Per aiutare gli sviluppatori a scrivere, testare ed eseguire il debug di questi resolver, la AWS AppSync console fornisce anche strumenti per creare una richiesta e una risposta GraphQL con dati fittizi fino al singolo resolver di campo. Inoltre, puoi eseguire query, mutazioni e sottoscrizioni nella AWS AppSync console e visualizzare un flusso di log dettagliato dell'intera richiesta proveniente da Amazon. CloudWatch Ciò include i risultati della fonte di dati.

Test con dati fittizi

Quando viene richiamato un resolver GraphQL, contiene un context oggetto che contiene informazioni pertinenti sulla richiesta. Tali informazioni includono gli argomenti provenienti da un client, le informazioni sull'identità e i dati del campo GraphQL padre. Memorizza anche i risultati della fonte di dati, che può essere utilizzata nel gestore delle risposte. Per ulteriori informazioni su questa struttura e sulle utilità di supporto disponibili da utilizzare durante la programmazione, vedete il riferimento all'oggetto contestuale [Resolver](#).

Quando si scrive o si modifica una funzione resolver, è possibile passare un oggetto contestuale fittizio o di test all'editor della console. Ciò consente di vedere come valutano sia il gestore di richiesta che quello di risposta senza che vengano effettivamente eseguiti su una fonte di dati. Puoi ad esempio passare un argomento firstname: Shaggy di test e vedere i relativi risultati quando usi `ctx.args.firstname` nel codice del modello. Puoi anche testare la valutazione di utilità helper, ad esempio `util.autoId()` o `util.time.nowISO8601()`.

Test dei resolver

Questo esempio utilizzerà la AWS AppSync console per testare i resolver.

1. [Accedi a Console di gestione AWS e apri la console. AppSync](#)
 - a. Nella APIs dashboard, scegli la tua API GraphQL.
 - b. Nella barra laterale, scegli Funzioni.
2. Scegli una funzione esistente.
3. Nella parte superiore della pagina Aggiorna funzione, scegli Seleziona contesto di test, quindi scegli Crea nuovo contesto.
4. Seleziona un oggetto di contesto di esempio o compila il codice JSON manualmente nella finestra Configura contesto di test di seguito.
5. Immettete un nome per il contesto di testo.
6. Seleziona il pulsante Save (Salva).
7. Per valutare il resolver utilizzando l'oggetto context fittizio, scegliere Run Test (Esegui test).

Per un esempio più pratico, supponiamo di avere un'app che memorizza un tipo GraphQL che utilizza la generazione automatica Dog di ID per gli oggetti e li archivia in Amazon DynamoDB. Vuoi anche scrivere alcuni valori dagli argomenti di una mutazione GraphQL e consentire solo a utenti specifici di vedere una risposta. Il seguente frammento mostra come potrebbe apparire lo schema:

```
type Dog {  
  breed: String  
  color: String  
}  
  
type Mutation {  
  addDog(firstname: String, age: Int): Dog  
}
```

Puoi scrivere una AWS AppSync funzione e aggiungerla al tuo addDog resolver per gestire la mutazione. Per testare la tua AWS AppSync funzione, puoi compilare un oggetto di contesto come nell'esempio seguente. Il seguente include gli argomenti name e age del client, oltre che un elemento username popolato nell'oggetto identity:

```
{  
  "arguments" : {  
    "firstname": "Shaggy",  
    "age": 4  
  },  
  "identity": {  
    "username": "Shaggy"  
  }  
}
```

```
"source" : {},
"result" : {
  "breed" : "Miniature Schnauzer",
  "color" : "black_grey"
},
"identity": {
  "sub" : "uuid",
  "issuer" : " https://cognito-idp.{region}.amazonaws.com/{userPoolId}",
  "username" : "Nadia",
  "claims" : { },
  "sourceIp" :[  "x.x.x.x" ],
  "defaultAuthStrategy" : "ALLOW"
}
}
```

È possibile testare la AWS AppSync funzione utilizzando il codice seguente:

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({ id: util.autoId() }),
    attributeValues: util.dynamodb.toMapValues(ctx.args),
  };
}

export function response(ctx) {
  if (ctx.identity.username === 'Nadia') {
    console.log("This request is allowed")
    return ctx.result;
  }
  util.unauthorized();
}
```

Il gestore di richieste e risposte valutato contiene i dati dell'oggetto del contesto di test e il valore generato da `util.autoId()`. Inoltre, se decidi di modificare `username` in un valore diverso da `Nadia`, i risultati non verranno restituiti perché il controllo di autorizzazione avrebbe esito negativo. [Per ulteriori informazioni sul controllo granulare degli accessi, consulta Casi d'uso delle autorizzazioni.](#)

Testare i gestori di richieste e risposte con's AWS AppSync APIs

Puoi utilizzare il comando EvaluateCode API per testare in remoto il tuo codice con dati simulati. Per iniziare con il comando, assicurati di aver aggiunto l'appsync:evaluateMappingCodeautorizzazione alla tua politica. Esempio:

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "appsync:evaluateCode",
      "Resource": "arn:aws:appsync:us-east-1:111122223333:*"
    }
  ]
}
```

È possibile sfruttare il comando utilizzando [AWS CLI](#) o [AWS SDKs](#). Ad esempio, prendiamo lo Dog schema e i relativi gestori di richiesta e risposta delle AWS AppSync funzioni della sezione precedente. Utilizzando la CLI sulla stazione locale, salvate il codice in un file denominato code.js, quindi salvate l'contextoggetto in un file denominato context.json Dalla tua shell, esegui il seguente comando:

```
$ aws appsync evaluate-code \
  --code file://code.js \
  --function response \
  --context file://context.json \
  --runtime name=APPSYNC_JS,runtimeVersion=1.0.0
```

La risposta contiene un file evaluationResult contenente il payload restituito dal gestore. Contiene anche un logs oggetto che contiene l'elenco dei log generati dal gestore durante la valutazione. Ciò semplifica il debug dell'esecuzione del codice e la visualizzazione delle informazioni sulla valutazione per facilitare la risoluzione dei problemi. Esempio:

```
{
  "evaluationResult": "{\"breed\":\"Miniature Schnauzer\",\"color\":\"black_grey\"}",
  "logs": [
```

```

    "INFO - code.js:13:5: \"This request is allowed\""
  ]
}

```

`evaluationResult` Può essere analizzato come JSON, il che fornisce:

```

{
  "breed": "Miniature Schnauzer",
  "color": "black_grey"
}

```

Utilizzando l'SDK, puoi incorporare facilmente i test della tua suite di test preferita per convalidare il comportamento dei tuoi gestori. Ti consigliamo di creare test utilizzando [Jest Testing Framework, ma qualsiasi suite di test](#) funziona. Il seguente frammento mostra un'ipotetica esecuzione di convalida. Nota che ci aspettiamo che la risposta di valutazione sia un codice JSON valido, quindi lo utilizziamo `JSON.parse` per recuperare JSON dalla stringa di risposta:

```

const AWS = require('aws-sdk')
const fs = require('fs')
const client = new AWS.AppSync({ region: 'us-east-2' })
const runtime = {name:'APPSYNC_JS',runtimeVersion:'1.0.0'}

test('request correctly calls DynamoDB', async () => {
  const code = fs.readFileSync('./code.js', 'utf8')
  const context = fs.readFileSync('./context.json', 'utf8')
  const contextJSON = JSON.parse(context)

  const response = await client.evaluateCode({ code, context, runtime, function:
'request' }).promise()
  const result = JSON.parse(response.evaluationResult)

  expect(result.key.id.S).toBeDefined()
  expect(result.attributeValues.firstname.S).toEqual(contextJSON.arguments.firstname)
})

```

Ciò produce il seguente risultato:

```

Ran all test suites.
> jest

PASS ./index.test.js

```



```
# request correctly calls DynamoDB (543 ms)
Test Suites: 1 passed, 1 total
Tests: 1 passed, 1 total
Snapshots: 0 totalTime: 1.511 s, estimated 2 s
```

Eseguire il debug di una query live

Non c'è nulla che possa sostituire un end-to-end test e una registrazione per eseguire il debug di un'applicazione di produzione. AWS AppSync consente di registrare gli errori e i dettagli completi delle richieste utilizzando Amazon CloudWatch. Inoltre, puoi utilizzare la AWS AppSync console per testare le query, le mutazioni e gli abbonamenti GraphQL e trasmettere in live streaming i dati di registro per ogni richiesta nell'editor di query per il debug in tempo reale. Per le sottoscrizioni, i log visualizzano le informazioni relative al tempo della connessione.

A tale scopo, è necessario che Amazon CloudWatch logs sia abilitato in anticipo, come descritto in [Monitoraggio e registrazione](#). Successivamente, nella AWS AppSync console, scegli la scheda Queries e inserisci una query GraphQL valida. Nella sezione in basso a destra, fai clic e trascina la finestra Logs per aprire la visualizzazione dei log. Sulla parte superiore della pagina, scegliere l'icona con la freccia per la riproduzione per eseguire la query GraphQL. In pochi istanti, i registri completi delle richieste e delle risposte relativi all'operazione vengono trasmessi in streaming a questa sezione e possono essere visualizzati nella console.

Configurazione e utilizzo dei resolver di pipeline in () AWS AppSync JavaScript

AWS AppSync esegue i resolver su un campo GraphQL. In alcuni casi, le applicazioni prevedono il compimento di più operazioni per la risoluzione di un singolo campo GraphQL. Con i resolver a pipeline, gli sviluppatori possono ora comporre operazioni chiamate Funzioni ed eseguirle in sequenza. Questo tipo di resolver torna utile, ad esempio, con applicazioni che prevedono un controllo delle autorizzazioni antecedente al recupero dei dati per un campo.

[Per ulteriori informazioni sull'architettura di un resolver di JavaScript pipeline, consulta la panoramica dei resolver. JavaScript](#)

Fase 1: Creazione di un risolutore di pipeline

Nella AWS AppSync console, vai alla pagina Schema.

Salva lo schema seguente:

```
schema {
```

```
    query: Query
    mutation: Mutation
  }

  type Mutation {
    signUp(input: Signup): User
  }

  type Query {
    getUser(id: ID!): User
  }

  input Signup {
    username: String!
    email: String!
  }

  type User {
    id: ID!
    username: String
    email: AWSEmail
  }
```

Bisogna implementare un resolver di pipeline per il campo `signUp` (registrazione) del tipo di `Mutation` (Mutazione). Nel tipo di mutazione sul lato destro, scegli `Allega` accanto al campo di `signUp` mutazione. Imposta il resolver su pipeline `resolver` e il `APPSYNC_JS` runtime, quindi crea il resolver.

Il nostro resolver di pipeline registra un utente convalidandone l'indirizzo e-mail e, successivamente, salvandolo nel sistema. Incapsuleremo la convalida dell'e-mail all'interno di una funzione `ValidateEmail` e il salvataggio dell'utente all'interno di una funzione `SaveUser`. Per prima, viene eseguita la funzione `validateEmail`, al termine della quale e solo se appurata la validità dell'e-mail, si può procedere con la `saveUser`.

Il flusso di esecuzione sarà il seguente:

1. Gestore di richieste del resolver `Mutation.signUp`
2. Funzione `validateEmail`
3. Funzione `saveUser`
4. Gestore di risposte del resolver `Mutation.signUp`

Poiché probabilmente riutilizzeremo la funzione `ValidateEmail` in altri resolver sulla nostra API, vogliamo evitare l'accesso `ctx.args` perché questi cambieranno da un campo GraphQL all'altro. In alternativa, è possibile avvalersi di `ctx.stash` per memorizzare l'attributo e-mail dall'argomento del campo di input `signUp(input: Signup)`.

Aggiorna il codice del resolver sostituendo le funzioni di richiesta e risposta:

```
export function request(ctx) {
  ctx.stash.email = ctx.args.input.email
  return {}
}

export function response(ctx) {
  return ctx.prev.result
}
```

Scegli Crea o Salva per aggiornare il resolver.

Fase 2: Creazione di una funzione

Dalla pagina Pipeline Resolver, nella sezione Funzioni, fai clic su **Aggiungi funzione**, quindi su **Crea nuova funzione**. È anche possibile creare funzioni senza passare dalla pagina del resolver; per farlo, nella AWS AppSync console, vai alla pagina Funzioni. Selezionare il pulsante **Create function** (Crea funzione). Creiamo quindi una funzione che verifichi la validità e la provenienza da un determinato dominio di un indirizzo e-mail. In caso di e-mail non valida, la funzione restituisce un errore. Altrimenti, inoltra qualsiasi input immesso.

Assicurati di aver creato una fonte di dati del tipo **NONE**. Scegli questa fonte di dati nell'elenco **Nome origine dati**. Per il nome della funzione, immettete `validateEmail`. Nell'area del codice della funzione, sovrascrivi tutto con questo frammento:

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { email } = ctx.stash;
  const valid = util.matches(
    '^[a-zA-Z0-9_+.-]+@((?:[a-zA-Z0-9-]+\\.)?[a-zA-Z]+\\.)(myvaliddomain)\\.com',
    email
  );
  if (!valid) {
    util.error(`"${email}" is not a valid email.`);
  }
}
```

```
    return { payload: { email } };  
  }  
  
  export function response(ctx) {  
    return ctx.result;  
  }  
}
```

Controlla i dati inseriti, quindi scegli Crea. A questo punto, la funzione `validateEmail` è stata creata. Ripeti questi passaggi per creare la funzione `SaveUser` con il codice seguente (per semplicità, utilizziamo una fonte di dati `NONE` e facciamo finta che l'utente sia stato salvato nel sistema dopo l'esecuzione della funzione.):

```
import { util } from '@aws-appsync/utils';  
  
export function request(ctx) {  
  return ctx.prev.result;  
}  
  
export function response(ctx) {  
  ctx.result.id = util.autoId();  
  return ctx.result;  
}
```

Abbiamo appena creato la nostra funzione `SaveUser`.

Fase 3: Aggiungere una funzione a un risolutore di pipeline

Le nostre funzioni avrebbero dovuto essere aggiunte automaticamente al risolutore di pipeline che abbiamo appena creato. Se così non fosse, o se hai creato le funzioni tramite la pagina Funzioni, puoi fare clic su **Aggiungi funzione** nuovamente nella pagina del `signUp` resolver per allegarle. Aggiungi entrambe le funzioni `ValidateEmail` e `SaveUser` al resolver. la funzione `validateEmail` deve precedere quella `saveUser`. Man mano che aggiungi altre funzioni, puoi utilizzare le opzioni di spostamento su e sposta giù per riorganizzare l'ordine di esecuzione delle funzioni. Controlla le modifiche, quindi scegli **Salva**.

Passaggio 4: Esecuzione di una query

Nella AWS AppSync console, vai alla pagina Query. Nell'explorer, assicurati di usare la tua mutazione. Se non lo sei, scegli `Mutation` nell'elenco a discesa, quindi scegli. **+** Inserire la query seguente:

```
mutation {
  signUp(input: {email: "nadia@myvaliddomain.com", username: "nadia"}) {
    id
    username
  }
}
```

Questo dovrebbe restituire qualcosa del tipo:

```
{
  "data": {
    "signUp": {
      "id": "256b6cc2-4694-46f4-a55e-8cb14cc5d7fc",
      "username": "nadia"
    }
  }
}
```

Abbiamo quindi registrato il nostro utente convalidandone, al contempo, l'e-mail di input tramite un resolver di pipeline.

Creazione di interrogazioni di base (VTL)

Note

Ora supportiamo principalmente il runtime APPSYNC_JS e la relativa documentazione. [Prendi in considerazione l'utilizzo del runtime APPSYNC_JS e delle relative guide qui.](#)

I resolver GraphQL connettono i campi nello schema di un tipo a un'origine dati. I resolver sono il meccanismo mediante il quale le richieste vengono soddisfatte. AWS AppSync possono creare e connettere automaticamente i resolver da uno schema oppure creare uno schema e connettere i resolver da una tabella esistente senza dover scrivere alcun codice.

Resolver AWS AppSync utilizzati JavaScript per convertire un'espressione GraphQL in un formato utilizzabile dall'origine dati. In alternativa, i modelli di mappatura possono essere scritti in [Apache Velocity Template Language \(VTL\)](#) per convertire un'espressione GraphQL in un formato utilizzabile dall'origine dati.

Questa sezione ti mostrerà come configurare i resolver usando VTL. [Una guida introduttiva alla programmazione in stile tutorial per la scrittura di resolver è disponibile nella guida alla programmazione dei modelli di mappatura Resolver](#), mentre le utilità di supporto disponibili per la programmazione sono disponibili nella guida al contesto del modello di mappatura Resolver. AWS AppSync dispone anche di flussi di test e debug integrati che puoi usare quando modifichi o crei da zero. Per ulteriori informazioni, consulta [Test and debug](#) resolvers.

Ti consigliamo di seguire questa guida prima di provare a utilizzare uno dei tutorial sopra menzionati.

In questa sezione, spiegheremo come creare un resolver, aggiungere un resolver per le mutazioni e utilizzare configurazioni avanzate.

Crea il tuo primo resolver

Seguendo gli esempi delle sezioni precedenti, il primo passo è creare un resolver adatto al tuo tipo.

Query

Console

1. [Accedi a Console di gestione AWS e apri la AppSync console.](#)
 - a. Nella APIs dashboard, scegli la tua API GraphQL.
 - b. Nella barra laterale, scegli Schema.
2. Sul lato destro della pagina, c'è una finestra chiamata Resolvers. Questa casella contiene un elenco dei tipi e dei campi definiti nella finestra Schema sul lato sinistro della pagina. È possibile allegare resolver ai campi. Ad esempio, nella sezione Tipo di query, scegli Allega accanto al campo. `getTodos`
3. Nella pagina Create Resolver, scegli l'origine dati che hai creato nella guida [Allegare una fonte di dati](#). Nella finestra Configura modelli di mappatura, puoi scegliere sia il modello generico di mappatura di richiesta che quello di risposta utilizzando l'elenco a discesa a destra o scriverne uno personalizzato.

 Note

L'associazione di un modello di mappatura delle richieste a un modello di mappatura delle risposte viene chiamata risolutore di unità. I resolver di unità sono in genere pensati per eseguire operazioni di routine; si consiglia di utilizzarli solo per operazioni singolari con un numero limitato di fonti di dati. Per operazioni più complesse,

consigliamo di utilizzare resolver a pipeline, che possono eseguire più operazioni con più fonti di dati in sequenza.

[Per ulteriori informazioni sulla differenza tra i modelli di mappatura delle richieste e delle risposte, consulta Unit resolvers.](#)

[Per ulteriori informazioni sull'utilizzo dei resolver pipeline, vedete Pipeline resolvers.](#)

4. Per i casi d'uso più comuni, la AWS AppSync console dispone di modelli integrati che è possibile utilizzare per recuperare elementi dalle fonti di dati (ad esempio, le query su tutti gli elementi, le ricerche individuali, ecc.). Ad esempio, nella versione semplice dello schema di [Designing your schema](#) in cui `getTodos` non c'era l'impaginazione, il modello di mappatura delle richieste per elencare gli articoli è il seguente:

```
{
  "version" : "2017-02-28",
  "operation" : "Scan"
}
```

5. È sempre necessario un modello di mappatura delle risposte da allegare alla richiesta. La console ne fornisce uno predefinito con il valore di `passthrough` seguente per elenchi:

```
$util.toJson($ctx.result.items)
```

In questo esempio, l'oggetto `context` (con alias `$ctx`) per gli elenchi di elementi presenta la forma `$context.result.items`. Se l'operazione GraphQL restituisce un singolo elemento, sarebbe `$context.result`. AWS AppSync fornisce funzioni di helper per operazioni comuni, ad esempio la funzione `$util.toJson` elencata in precedenza, per formattare le risposte correttamente. Per un elenco completo delle funzioni, consulta il riferimento all'utilità del modello di [mappatura Resolver](#).

6. Scegli `Save Resolver`.

API

1. Crea un oggetto resolver chiamando l'API. [CreateResolver](#)
2. Puoi modificare i campi del tuo resolver chiamando l'API. [UpdateResolver](#)

CLI

1. Crea un resolver eseguendo il comando. [create-resolver](#)

Dovrai digitare 6 parametri per questo particolare comando:

1. La `api-id` della tua API.
2. Il `type-name` tipo che desideri modificare nel tuo schema. Nell'esempio della console, questo era `Query`.
3. Il `field-name` campo che vuoi modificare nel tuo tipo. Nell'esempio della console, questo era `getTodos`.
4. La fonte `data-source-name` di dati che hai creato nella guida [Allegare una fonte di dati](#).
5. Il `request-mapping-template`, che è il corpo della richiesta. Nell'esempio della console, questo era:

```
{
  "version" : "2017-02-28",
  "operation" : "Scan"
}
```

6. Il `response-mapping-template`, che è il corpo della risposta. Nell'esempio della console, questo era:

```
$util.toJson($ctx.result.items)
```

Un comando di esempio può essere simile al seguente:

```
aws appsync create-resolver --api-id abcdefghijklmnopqrstuvwxyz --type-name
Query --field-name getTodos --data-source-name TodoTable --request-mapping-
template "{ \"version\" : \"2017-02-28\", \"operation\" : \"Scan\", }" --response-
mapping-template "\"$util.toJson('$ctx.result.items')"
```

Un output verrà restituito nella CLI. Ecco un esempio:

```
{
  "resolver": {
    "kind": "UNIT",
    "dataSourceName": "TodoTable",
```



```

    "requestMappingTemplate": "{ version : 2017-02-28, operation : Scan, }",
    "resolverArn": "arn:aws:appsync:us-west-2:107289374856:apis/
    abcdefghijklmnopqrstuvwxyz/types/Query/resolvers/getTodos",
    "typeName": "Query",
    "fieldName": "getTodos",
    "responseMappingTemplate": "$util.toJson($ctx.result.items)"
  }
}

```

2. Per modificare i modelli di and/or mappatura dei campi di un resolver, esegui il comando. [update-resolver](#)

Ad eccezione del `api-id` parametro, i parametri utilizzati nel `create-resolver` comando verranno sovrascritti dai nuovi valori del comando. `update-resolver`

Aggiungere un resolver per le mutazioni

Il passaggio successivo consiste nel creare un resolver adatto al tuo tipo. `Mutation`

Console

1. [Accedi a Console di gestione AWS e apri la AppSync console.](#)
 - a. Nella APIs dashboard, scegli la tua API GraphQL.
 - b. Nella barra laterale, scegli Schema.
2. Nella sezione Tipo di mutazione, scegli Allega accanto al `addTodo` campo.
3. Nella pagina Create Resolver, scegli l'origine dati che hai creato nella guida [Allegare una fonte di dati](#).
4. Nella finestra Configura modelli di mappatura, è necessario modificare il modello di richiesta perché si tratta di una mutazione in cui si aggiunge un nuovo elemento a DynamoDB. Usa modello di mappatura della richiesta seguente:

```

{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key" : {
    "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
  },
  "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}

```

5. AWS AppSync converte automaticamente gli argomenti definiti nel `addTodo` campo dallo schema GraphQL in operazioni DynamoDB. L'esempio precedente archivia i record in DynamoDB utilizzando una chiave `id` `of`, che viene passata dall'argomento di mutazione `as.$ctx.args.id`. Tutti gli altri campi che passi attraverso vengono mappati automaticamente agli attributi DynamoDB con `$util.dynamodb.toMapValuesJson($ctx.args)`.

Per questo resolver, usare il seguente modello di mappatura della risposta:

```
$util.toJson($ctx.result)
```

AWS AppSync supporta anche flussi di lavoro di test e debug per la modifica dei resolver. È possibile utilizzare un oggetto `context` fittizio per visualizzare il valore trasformato del modello prima di effettuare la chiamata. Eventualmente, è possibile visualizzare l'esecuzione di richiesta completa a un'origine dati in modo interattivo quando si esegue una query. [Per ulteriori informazioni, consulta Test e debug resolvers e Monitoraggio e registrazione.](#)

6. Scegli `Save Resolver`.

API

Puoi farlo anche APIs utilizzando i comandi nella sezione [Crea il tuo primo resolver](#) e i dettagli dei parametri in questa sezione.

CLI

Puoi farlo anche nella CLI utilizzando i comandi nella sezione [Crea il tuo primo resolver](#) e i dettagli dei parametri in questa sezione.

[A questo punto, se non utilizzi i resolver avanzati, puoi iniziare a utilizzare l'API GraphQL come descritto in Utilizzo dell'API.](#)

Resolver avanzati

Se stai seguendo la sezione Avanzate e stai creando uno schema di esempio in [Progettazione dello schema](#) per eseguire una scansione impaginata, usa invece il seguente modello di richiesta per il campo: `getTodos`

```
{
  "version" : "2017-02-28",
  "operation" : "Scan",
```

```
"limit": $util.defaultIfNull(${ctx.args.limit}, 20),  
"nextToken": $util.toJson($util.defaultIfNullOrBlank($ctx.args.nextToken, null))  
}
```

Per questo caso d'uso della paginazione, la mappatura della risposta è più di un semplice passthrough perché deve contenere sia il cursore (in modo che il client sappia a quale pagina passare) che il set di risultati. Il modello di mappatura è come segue:

```
{  
  "todos": $util.toJson($context.result.items),  
  "nextToken": $util.toJson($context.result.nextToken)  
}
```

I campi nel modello di mappatura della risposta precedente devono corrispondere ai campi definiti nel tipo `TodoConnection`.

Nel caso di relazioni in cui si dispone di una `Comments` tabella e si sta risolvendo il campo dei commenti sul `Todo` tipo (che restituisce un tipo di `[Comment]`), è possibile utilizzare un modello di mappatura che esegue una query sulla seconda tabella. A tale scopo, è necessario aver già creato un'origine dati per la `Comments` tabella, come descritto in [Allegare](#) un'origine dati.

Note

Stiamo utilizzando un'operazione di interrogazione su una seconda tabella solo a scopo illustrativo. È possibile utilizzare invece un'altra operazione su DynamoDB. Inoltre, puoi estrarre i dati da un'altra fonte di dati, ad AWS Lambda esempio Amazon OpenSearch Service, perché la relazione è controllata dal tuo schema GraphQL.

Console

1. Accedi a Console di gestione AWS e apri la [AppSync console](#).
 - a. Nella APIs dashboard, scegli la tua API GraphQL.
 - b. Nella barra laterale, scegli Schema.
2. Nel tipo `Todo`, scegli `Allega` accanto al `comments` campo.
3. Nella pagina `Create Resolver`, scegli l'origine dati della tabella `Commenti`. Il nome predefinito per la tabella `Commenti` delle guide di avvio rapido è `AppSyncCommentTable`, ma può variare a seconda del nome assegnato.

4. Aggiungi il seguente frammento al modello di mappatura della richiesta:

```
{
  "version": "2017-02-28",
  "operation": "Query",
  "index": "todoid-index",
  "query": {
    "expression": "todoid = :todoid",
    "expressionValues": {
      ":todoid": {
        "S": $util.toJson($context.source.id)
      }
    }
  }
}
```

5. `context.source` fa riferimento all'oggetto padre del campo corrente che viene risolto. In questo esempio, `source.id` si riferisce al singolo oggetto `Todo`, che viene quindi utilizzato per l'espressione di query.

Puoi usare il modello di mappatura della risposta passthrough come segue:

```
$util.toJson($ctx.result.items)
```

6. Scegli `Save Resolver`.
7. Infine, torna alla pagina Schema della console, collega un resolver al `addComment` campo e specifica l'origine dati per la tabella. `Comments` Il modello di mappatura della richiesta in questo caso è un semplice oggetto `PutItem` con il `todoid` specifico commentato su un argomento, ma puoi utilizzare l'utilità `$utils.autoId()` per creare una chiave di ordinamento per il commento, come segue:

```
{
  "version": "2017-02-28",
  "operation": "PutItem",
  "key": {
    "todoid": { "S": $util.toJson($context.arguments.todoid) },
    "commentid": { "S": "$util.autoId()" }
  },
  "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}
```

Usa un modello di risposta passthrough come segue:

```
$util.toJson($ctx.result)
```

API

Puoi farlo anche APIs utilizzando i comandi nella sezione [Crea il tuo primo resolver](#) e i dettagli dei parametri in questa sezione.

CLI

Puoi farlo anche nella CLI utilizzando i comandi nella sezione [Crea il tuo primo resolver](#) e i dettagli dei parametri in questa sezione.

Disabilitazione dei modelli di mappatura VTL con resolver Lambda diretti (VTL)

Note

Ora supportiamo principalmente il runtime APPSYNC_JS e la relativa documentazione. [Prendi in considerazione l'utilizzo del runtime APPSYNC_JS e delle relative guide qui.](#)

Con i resolver diretti Lambda, puoi aggirare l'uso di modelli di mappatura VTL quando utilizzi fonti di dati. AWS Lambda AWS AppSync può fornire un payload predefinito alla funzione Lambda e una traduzione predefinita dalla risposta di una funzione Lambda a un tipo GraphQL. Puoi scegliere di fornire un modello di richiesta, un modello di risposta o nessuno dei due e lo AWS AppSync gestirà di conseguenza.

Per ulteriori informazioni sul payload predefinito della richiesta e sulla traduzione delle risposte che AWS AppSync fornisce, consulta il riferimento al [resolver Direct Lambda](#). Per ulteriori informazioni sulla configurazione di un'origine AWS Lambda dati e sulla configurazione di una IAM Trust Policy, consulta [Allegare](#) un'origine dati.

Configurazione di resolver Lambda diretti

Le seguenti sezioni ti mostreranno come collegare sorgenti dati Lambda e aggiungere resolver Lambda ai tuoi campi.

Aggiungere un'origine dati Lambda

Prima di poter attivare i resolver Lambda diretti, devi aggiungere un'origine dati Lambda.

Console

1. [Accedi e apri la console. Console di gestione AWS AppSync](#)
 - a. Nella APIs dashboard, scegli la tua API GraphQL.
 - b. Nella barra laterale, scegli Origini dati.
2. Seleziona Create data source (Crea origine dati).
 - a. Per Nome dell'origine dati, inserisci un nome per la tua origine dati, ad esempio **myFunction**.
 - b. Per Tipo di origine dati, scegli AWS Lambda funzione.
 - c. Per Regione, scegli la regione appropriata.
 - d. Per Funzione ARN, scegli la funzione Lambda dall'elenco a discesa. È possibile cercare il nome della funzione o inserire manualmente l'ARN della funzione che si desidera utilizzare.
 - e. Crea un nuovo ruolo IAM (consigliato) o scegli un ruolo esistente con l'autorizzazione `lambda:invokeFunction` IAM. I ruoli esistenti richiedono una policy di fiducia, come spiegato nella sezione [Allegare una fonte di dati](#).

Di seguito è riportato un esempio di policy IAM che dispone delle autorizzazioni necessarie per eseguire operazioni sulla risorsa:

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [ "lambda:invokeFunction" ],
      "Resource": [
        "arn:aws:lambda:us-west-2:123456789012:function:myFunction",
        "arn:aws:lambda:us-west-2:123456789012:function:myFunction:*"
      ]
    }
  ]
}
```

```

    ]
  }
}

```

- Scegli il pulsante Crea.

CLI

- Crea un oggetto sorgente dati eseguendo il [create-data-source](#) comando.

Dovrai digitare 4 parametri per questo particolare comando:

- La `api-id` della tua API.
- La tua fonte `name` di dati. Nell'esempio della console, questo è il nome dell'origine dati.
- La fonte `type` dei dati. Nell'esempio della console, questa è AWS Lambda la funzione.
- Il `lambda-config`, che è l'ARN della funzione nell'esempio della console.

Note

Esistono altri parametri come questi `Region` che devono essere configurati, ma di solito vengono utilizzati per impostazione predefinita i valori di configurazione CLI.

Un comando di esempio può avere il seguente aspetto:

```
aws appsync create-data-source --api-id abcdefghijklmnopqrstuvwxyz
--name myFunction --type AWS_LAMBDA --lambda-config
lambdaFunctionArn=arn:aws:lambda:us-west-2:102847592837:function:appsync-
lambda-example
```

Un output verrà restituito nella CLI. Ecco un esempio:

```
{
  "dataSource": {
    "dataSourceArn": "arn:aws:appsync:us-west-2:102847592837:apis/
abcdefghijklmnopqrstuvwxyz/datasources/myFunction",
    "type": "AWS_LAMBDA",
    "name": "myFunction",
```

```
"lambdaConfig": {
  "lambdaFunctionArn": "arn:aws:lambda:us-
west-2:102847592837:function:appsync-lambda-example"
}
}
```

2. Per modificare gli attributi di un'origine dati, esegui il [update-data-source](#) comando.

Ad eccezione del `api-id` parametro, i parametri utilizzati nel `create-data-source` comando verranno sovrascritti dai nuovi valori del `update-data-source` comando.

Attiva i resolver Lambda diretti

Dopo aver creato un'origine dati Lambda e impostato il ruolo IAM appropriato per consentire di AWS AppSync richiamare la funzione, puoi collegarla a una funzione resolver o pipeline.

Console

1. [Accedi a e apri la console. Console di gestione AWS AppSync](#)
 - a. Nella APIs dashboard, scegli la tua API GraphQL.
 - b. Nella barra laterale, scegli Schema.
2. Nella finestra Resolver, scegli un campo o un'operazione, quindi seleziona il pulsante Allega.
3. Nella pagina Crea nuovo resolver, scegli la funzione Lambda dall'elenco a discesa.
4. Per sfruttare i resolver diretti Lambda, verifica che i modelli di mappatura di richiesta e risposta siano disabilitati nella sezione Configura modelli di mappatura.
5. Scegli il pulsante Save Resolver.

CLI

- Crea un resolver eseguendo il comando. [create-resolver](#)

Dovrai digitare 6 parametri per questo particolare comando:

1. La `api-id` della tua API.
2. Il `type-name` tipo nel tuo schema.
3. Il campo `field-name` del tuo schema.

4. `ldata-source-name`, o il nome della tua funzione Lambda.
5. `lrequest-mapping-template`, che è il corpo della richiesta. Nell'esempio della console, questo era disabilitato:

```
" "
```

6. `lresponse-mapping-template`, che è il corpo della risposta. Nell'esempio della console, anche questo era disabilitato:

```
" "
```

Un comando di esempio può avere il seguente aspetto:

```
aws appsync create-resolver --api-id abcdefghijklmnopqrstuvwxyz --type-name  
Subscription --field-name onCreateTodo --data-source-name LambdaTest --request-  
mapping-template " " --response-mapping-template " "
```

Un output verrà restituito nella CLI. Ecco un esempio:

```
{  
  "resolver": {  
    "resolverArn": "arn:aws:appsync:us-west-2:102847592837:apis/  
abcdefghijklmnopqrstuvwxyz/types/Subscription/resolvers/onCreateTodo",  
    "typeName": "Subscription",  
    "kind": "UNIT",  
    "fieldName": "onCreateTodo",  
    "dataSourceName": "LambdaTest"  
  }  
}
```

Quando disabiliti i modelli di mappatura, si verificheranno diversi comportamenti aggiuntivi in: AWS AppSync

- Disabilitando un modello di mappatura, stai segnalando AWS AppSync che accetti le traduzioni dei dati predefinite specificate nel riferimento del resolver Direct [Lambda](#).
- [Disabilitando il modello di mappatura della richiesta, l'origine dati Lambda riceverà un payload costituito dall'intero oggetto Context.](#)

- Disabilitando il modello di mappatura delle risposte, il risultato della chiamata Lambda verrà tradotto a seconda della versione del modello di mappatura della richiesta o se anche il modello di mappatura della richiesta è disabilitato.

Test e debug dei resolver in (VTL) AWS AppSync

Note

Ora supportiamo principalmente il runtime APPSYNC_JS e la relativa documentazione. [Prendi in considerazione l'utilizzo del runtime APPSYNC_JS e delle relative guide qui.](#)

AWS AppSync esegue resolver su un campo GraphQL su un'origine dati. Come descritto nella [panoramica dei modelli di mappatura Resolver](#), i resolver comunicano con le fonti di dati utilizzando un linguaggio di template. Ciò consente di personalizzare il comportamento e applicare logica e condizioni prima e dopo la comunicazione con l'origine dati. [Per una guida introduttiva alla programmazione in stile tutorial per la scrittura di resolver, consulta la guida alla programmazione dei modelli di mappatura Resolver.](#)

Per aiutare gli sviluppatori a scrivere, testare ed eseguire il debug di questi resolver, la AWS AppSync console fornisce anche strumenti per creare una richiesta e una risposta GraphQL con dati fittizi fino al singolo resolver di campo. Inoltre, puoi eseguire query, mutazioni e sottoscrizioni nella AWS AppSync console e visualizzare un flusso di log dettagliato da Amazon CloudWatch dell'intera richiesta. Ciò include i risultati di un'origine dati.

Test con dati fittizi

Quando viene richiamato un resolver GraphQL, contiene un context oggetto che contiene informazioni sulla richiesta. Tali informazioni includono gli argomenti provenienti da un client, le informazioni sull'identità e i dati del campo GraphQL padre. Contiene anche i risultati della fonte di dati, che possono essere utilizzati nel modello di risposta. Per ulteriori informazioni su questa struttura e sulle utilità helper disponibili per la programmazione, consulta le [informazioni di riferimento contestuali sui modelli di mappatura dei resolver](#).

Quando si scrive o si modifica un resolver, è possibile passare un oggetto di contesto fittizio o di test all'editor della console. In questo modo è possibile vedere in che modo i modelli di richiesta e di risposta eseguono la valutazione senza effettivamente eseguire un'origine dati. Puoi ad esempio passare un argomento `firstname: Shaggy` di test e vedere i relativi risultati quando usi

`$ctx.args.firstname` nel codice del modello. Puoi anche testare la valutazione di utilità helper, ad esempio `$util.autoId()` o `util.time.nowISO8601()`.

Test dei resolver

Questo esempio utilizzerà la AWS AppSync console per testare i resolver.

1. [Accedi a Console di gestione AWS e apri la console. AppSync](#)
 - a. Nella APIs dashboard, scegli la tua API GraphQL.
 - b. Nella barra laterale, scegli Schema.
2. Se non l'hai già fatto, sotto il tipo e accanto al campo, scegli Allega per aggiungere il tuo resolver.

[Per ulteriori informazioni su come creare un resolver completo, consulta Configurazione dei resolver.](#)

Altrimenti, seleziona il resolver già presente nel campo.

3. Nella parte superiore della pagina Modifica resolver, scegli Seleziona contesto di test, scegli Crea nuovo contesto.
4. Seleziona un oggetto contestuale di esempio o compila il JSON manualmente nella finestra del contesto di esecuzione sottostante.
5. Immettete un nome di contesto di testo.
6. Seleziona il pulsante Save (Salva).
7. Nella parte superiore della pagina Edit Resolver, scegli Esegui test.

Per un esempio più pratico, supponiamo di avere un'app che memorizza un tipo GraphQL che utilizza la generazione automatica Dog di ID per gli oggetti e li archivia in Amazon DynamoDB. Devi inoltre scrivere alcuni valori dagli argomenti di una mutazione GraphQL e permettere la visualizzazione di una risposta solo a determinati utenti. Di seguito viene mostrato come potrebbe apparire lo schema:

```
type Dog {  
  breed: String  
  color: String  
}  
  
type Mutation {  
  addDog(firstname: String, age: Int): Dog  
}
```

Quando aggiungi un resolver per la addDog mutazione, puoi popolare un oggetto di contesto come nell'esempio seguente. Il seguente include gli argomenti name e age del client, oltre che un elemento username popolato nell'oggetto identity:

```
{
  "arguments" : {
    "firstname": "Shaggy",
    "age": 4
  },
  "source" : {},
  "result" : {
    "breed" : "Miniature Schnauzer",
    "color" : "black_grey"
  },
  "identity": {
    "sub" : "uuid",
    "issuer" : " https://cognito-idp.{region}.amazonaws.com/{userPoolId}",
    "username" : "Nadia",
    "claims" : { },
    "sourceIp" :[ "x.x.x.x" ],
    "defaultAuthStrategy" : "ALLOW"
  }
}
```

Puoi eseguire un test usando i modelli di mappatura della richiesta e della risposta seguenti:

Modello di richiesta

```
{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key" : {
    "id" : { "S" : "$util.autoId()" }
  },
  "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}
```

Modello di risposta

```
#if ($context.identity.username == "Nadia")
  $util.toJson($ctx.result)
#else
```

```
$util.unauthorized()  
#end
```

Il modello valutato contiene i dati dell'oggetto context di test e il valore generato da `$util.autoId()`. Inoltre, se decidi di modificare username in un valore diverso da Nadia, i risultati non verranno restituiti perché il controllo di autorizzazione avrebbe esito negativo. [Per ulteriori informazioni sul controllo granulare degli accessi, consulta Casi d'uso delle autorizzazioni.](#)

Test dei modelli di mappatura con's AWS AppSync APIs

Puoi utilizzare il comando `EvaluateMappingTemplate` API per testare in remoto i tuoi modelli di mappatura con dati simulati. Per iniziare con il comando, assicurati di aver aggiunto l'`appsync:evaluateMappingTemplate` autorizzazione alla tua politica. Esempio:

JSON

```
{  
  "Version": "2012-10-17",  
  "Statement": [  
    {  
      "Effect": "Allow",  
      "Action": "appsync:evaluateMappingTemplate",  
      "Resource": "arn:aws:appsync:us-east-1:111122223333:*"  
    }  
  ]  
}
```

È possibile sfruttare il comando utilizzando [AWS CLI](#) o [AWS SDKs](#). Ad esempio, prendiamo lo Dog schema e i relativi modelli di request/response mappatura della sezione precedente. Utilizzando la CLI sulla stazione locale, salvate il modello di richiesta in un file denominato `request.vtl`, quindi salvate l'oggetto context in un file denominato `context.json`. Dalla tua shell, esegui il seguente comando:

```
aws appsync evaluate-mapping-template --template file://request.vtl --context file://  
context.json
```

Il comando restituisce la seguente risposta:

```
{
```

```
"evaluationResult": "{\n  \"version\" : \"2017-02-28\",\n  \"operation\" : \"PutItem\",\n  \"key\" : {\n    \"id\" : { \"S\" :\n      \"afcb4c85-49f8-40de-8f2b-248949176456\" }\n  },\n  \"attributeValues\" :\n    {\n      \"firstname\" : { \"S\" : \"Shaggy\" },\n      \"age\" : { \"N\" : 4 }\n    }\n  }\n}
```

`evaluationResult` contiene i risultati del test del modello fornito con quello fornito `context`. Puoi anche testare i tuoi modelli utilizzando il AWS SDKs. Ecco un esempio di utilizzo dell' AWS SDK per la versione 2 JavaScript :

```
const AWS = require('aws-sdk')
const client = new AWS.AppSync({ region: 'us-east-2' })

const template = fs.readFileSync('./request.vtl', 'utf8')
const context = fs.readFileSync('./context.json', 'utf8')

client
  .evaluateMappingTemplate({ template, context })
  .promise()
  .then((data) => console.log(data))
```

Utilizzando l'SDK, puoi incorporare facilmente i test della tua suite di test preferita per convalidare il comportamento del tuo modello. Ti consigliamo di creare test utilizzando [Jest Testing Framework](#), ma qualsiasi suite di test funziona. Il seguente frammento mostra un'ipotetica esecuzione di convalida. Nota che ci aspettiamo che la risposta di valutazione sia un codice JSON valido, quindi lo utilizziamo `JSON.parse` per recuperare JSON dalla stringa di risposta:

```
const AWS = require('aws-sdk')
const fs = require('fs')
const client = new AWS.AppSync({ region: 'us-east-2' })

test('request correctly calls DynamoDB', async () => {
  const template = fs.readFileSync('./request.vtl', 'utf8')
  const context = fs.readFileSync('./context.json', 'utf8')
  const contextJSON = JSON.parse(context)

  const response = await client.evaluateMappingTemplate({ template,
    context }).promise()
  const result = JSON.parse(response.evaluationResult)

  expect(result.key.id.S).toBeDefined()
  expect(result.attributeValues.firstname.S).toEqual(contextJSON.arguments.firstname)
```

```
})
```

Ciò produce il seguente risultato:

```
Ran all test suites.  
> jest  
  
PASS ./index.test.js  
# request correctly calls DynamoDB (543 ms)  
  
Test Suites: 1 passed, 1 total  
Tests: 1 passed, 1 total  
Snapshots: 0 total  
Time: 1.511 s, estimated 2 s
```

Eseguire il debug di una query live

Non c'è nulla che possa sostituire un end-to-end test e una registrazione per eseguire il debug di un'applicazione di produzione. AWS AppSync consente di registrare gli errori e i dettagli completi delle richieste utilizzando Amazon CloudWatch. Inoltre, puoi utilizzare la AWS AppSync console per testare le query, le mutazioni e gli abbonamenti GraphQL e trasmettere in live streaming i dati di registro per ogni richiesta nell'editor di query per il debug in tempo reale. Per le sottoscrizioni, i log visualizzano le informazioni relative al tempo della connessione.

A tale scopo, è necessario che Amazon CloudWatch logs sia abilitato in anticipo, come descritto in [Monitoraggio e registrazione](#). Successivamente, nella AWS AppSync console, scegli la scheda Queries e inserisci una query GraphQL valida. Nella sezione in basso a destra, fai clic e trascina la finestra Logs per aprire la visualizzazione dei log. Sulla parte superiore della pagina, scegliere l'icona con la freccia per la riproduzione per eseguire la query GraphQL. Dopo alcuni istati, i log completi per la richiesta e la risposta per l'operazione verranno trasmessi in questa sezione della console, dove potrai visualizzarli.

Configurazione e utilizzo dei resolver di pipeline in (VTL) AWS AppSync

Note

Ora supportiamo principalmente il runtime APPSYNC_JS e la relativa documentazione. [Prendi in considerazione l'utilizzo del runtime APPSYNC_JS e delle relative guide qui.](#)

AWS AppSync esegue i resolver su un campo GraphQL. In alcuni casi, le applicazioni prevedono il compimento di più operazioni per la risoluzione di un singolo campo GraphQL. Con i resolver a pipeline, gli sviluppatori possono ora comporre operazioni chiamate Funzioni ed eseguirle in sequenza. Questo tipo di resolver torna utile, ad esempio, con applicazioni che prevedono un controllo delle autorizzazioni antecedente al recupero dei dati per un campo.

Un resolver di pipeline è composto da due modelli di mappatura, della fase antecedente e successiva, e un elenco di funzioni. Ogni funzione dispone di un modello di mappatura delle richieste e delle risposte che esegue su un'origine dati. Dal momento che delega l'esecuzione a un elenco di funzioni, il resolver di pipeline non prevede il collegamento a un'origine dati. I resolver e le funzioni di unità sono primitive che eseguono operazioni su fonti di dati. Per ulteriori informazioni, consulta la panoramica del [modello di mappatura Resolver](#).

Fase 1: Creazione di un risolutore di pipeline

Nella AWS AppSync console, vai alla pagina Schema.

Salva lo schema seguente:

```
schema {
  query: Query
  mutation: Mutation
}

type Mutation {
  signUp(input: Signup): User
}

type Query {
  getUser(id: ID!): User
}

input Signup {
  username: String!
  email: String!
}

type User {
  id: ID!
  username: String
  email: AWSEmail
}
```


Bisogna implementare un resolver di pipeline per il campo `signUp` (registrazione) del tipo di `Mutation` (Mutazione). Nel tipo di mutazione sul lato destro, scegli `Allega` accanto al campo di `signUp` mutazione. Nella pagina di creazione del resolver, fai clic su `Azioni`, quindi su `Aggiorna runtime`. Scegli **Pipeline Resolver**, quindi scegli **VTL**, quindi scegli `Aggiorna`. La pagina dovrebbe ora mostrare tre sezioni: un'area di testo `Prima della mappatura del modello`, una sezione `Funzioni` e un'area di testo `Dopo la mappatura del modello`.

Il nostro resolver di pipeline registra un utente convalidandone l'indirizzo e-mail e, successivamente, salvandolo nel sistema. Dobbiamo quindi incapsulare la convalida dell'e-mail in una funzione `validateEmail` e il salvataggio dell'utente in una funzione `saveUser`. Per prima, viene eseguita la funzione `validateEmail`, al termine della quale è solo se appurata la validità dell'e-mail, si può procedere con la `saveUser`.

Il flusso di esecuzione corrisponderà al seguente:

1. Modello di mappatura della richiesta del resolver `Mutation.signUp`
2. Funzione `validateEmail`
3. Funzione `saveUser`
4. Modello di mappatura della risposta del resolver `Mutation.signUp`

Poiché probabilmente riutilizzeremo la funzione `ValidateEmail` in altri resolver sulla nostra API, vogliamo evitare l'accesso `$ctx.args` perché questi cambieranno da un campo GraphQL all'altro. In alternativa, è possibile avvalersi di `$ctx.stash` per memorizzare l'attributo e-mail dall'argomento del campo di input `signUp(input: Signup)`.

PRIMA di mappare il modello:

```
## store email input field into a generic email key
$util.qr($ctx.stash.put("email", $ctx.args.input.email))
{}
```

La console fornisce un modello di mappatura `AFTER passthrough` predefinito che utilizzeremo:

```
$util.toJson($ctx.result)
```

Scegli `Crea` o `Salva` per aggiornare il resolver.

Fase 2: Creazione di una funzione

Dalla pagina Pipeline Resolver, nella sezione Funzioni, fai clic su **Aggiungi funzione**, quindi su **Crea nuova funzione**. È anche possibile creare funzioni senza passare dalla pagina del resolver; per farlo, nella AWS AppSync console, vai alla pagina Funzioni. Selezionare il pulsante **Create function** (Crea funzione). Creiamo quindi una funzione che verifichi la validità e la provenienza da un determinato dominio di un indirizzo e-mail. In caso di e-mail non valida, la funzione restituisce un errore. Altrimenti, inoltra qualsiasi input immesso.

Nella pagina della nuova funzione, scegli **Azioni**, quindi **Aggiorna runtime**. Scegli **VTL**, quindi **Aggiorna**. Assicurati di aver creato un'origine dati del tipo **NONE**. Scegli questa fonte di dati nell'elenco **Nome origine dati**. Per il nome della funzione, inserisci `validateEmail`. Nell'area del codice della funzione, sovrascrivi tutto con questo frammento:

```
#set($valid = $util.matches("^[a-zA-Z0-9_+-.]+@((?:[a-zA-Z0-9-]+\.)?[a-zA-Z]+\.)?(myvaliddomain)\.com", $ctx.stash.email))
#if (!$valid)
    $util.error("$ctx.stash.email is not a valid email.")
#end
{
    "payload": { "email": $util.toJson($ctx.stash.email) }
}
```

Incollalo nel modello di mappatura delle risposte:

```
$util.toJson($ctx.result)
```

Controlla le modifiche, quindi scegli **Crea**. A questo punto, la funzione `validateEmail` è stata creata. Ripeti questi passaggi per creare la funzione `SaveUser` con i seguenti modelli di mappatura di richieste e risposte (per semplicità, utilizziamo una fonte di dati **NONE** e facciamo finta che l'utente sia stato salvato nel sistema dopo l'esecuzione della funzione.):

Modello di mappatura della richiesta:

```
## $ctx.prev.result contains the signup input values. We could have also
## used $ctx.args.input.
{
    "payload": $util.toJson($ctx.prev.result)
}
```

Modello di mappatura della risposta:

```
## an id is required so let's add a unique random identifier to the output
$util.qr($ctx.result.put("id", $util.autoId()))
$util.toJson($ctx.result)
```

Abbiamo appena creato la nostra funzione SaveUser.

Fase 3: Aggiungere una funzione a un risolutore di pipeline

Le nostre funzioni avrebbero dovuto essere aggiunte automaticamente al risolutore di pipeline che abbiamo appena creato. Se così non fosse, o se hai creato le funzioni tramite la pagina Funzioni, puoi fare clic su **Aggiungi funzione** nella pagina del resolver per allegarle. Aggiungi entrambe le funzioni `ValidateEmail` e `SaveUser` al resolver. la funzione `validateEmail` deve precedere quella `saveUser`. Man mano che aggiungi altre funzioni, puoi utilizzare le opzioni di spostamento su e sposta giù per riorganizzare l'ordine di esecuzione delle funzioni. Controlla le modifiche, quindi scegli **Salva**.

Fase 4: Esecuzione di una query

Nella AWS AppSync console, vai alla pagina Query. Nell'explorer, assicurati di usare la tua mutazione. Se non lo sei, scegli `Mutation` nell'elenco a discesa, quindi scegli. **+** Inserire la query seguente:

```
mutation {
  signUp(input: {
    email: "nadia@myvaliddomain.com"
    username: "nadia"
  }) {
    id
    email
  }
}
```

Questo dovrebbe restituire qualcosa del tipo:

```
{
  "data": {
    "signUp": {
      "id": "256b6cc2-4694-46f4-a55e-8cb14cc5d7fc",
      "email": "nadia@myvaliddomain.com"
    }
  }
}
```

```
}  
}  
}
```

Abbiamo quindi registrato il nostro utente convalidandone, al contempo, l'e-mail di input tramite un resolver di pipeline. Un tutorial più articolato sui resolver di pipeline è disponibile alla pagina [Tutorial: resolver di pipeline](#).

Utilizzo di un' AWS AppSync API con AWS CDK

Tip

Prima di utilizzare il CDK, ti consigliamo di consultare la [documentazione ufficiale del CDK insieme AWS AppSync al riferimento CDK](#).

Ti consigliamo inoltre di assicurarti che le installazioni [AWS CLI](#) e [NPM](#) funzionino sul tuo sistema.

In questa sezione, creeremo una semplice applicazione CDK in grado di aggiungere e recuperare elementi da una tabella DynamoDB. [Questo vuole essere un esempio di avvio rapido che utilizza parte del codice delle sezioni Progettazione dello schema, Collegamento di una fonte di dati e Configurazione dei resolvers \(\)](#). JavaScript

Configurazione di un progetto CDK

Warning

Questi passaggi potrebbero non essere completamente accurati a seconda dell'ambiente. Partiamo dal presupposto che sul sistema siano installate le utilità necessarie, un modo per interfacciarsi con AWS i servizi e configurazioni corrette.

Il primo passo è l'installazione del CDK. AWS Nella CLI, puoi inserire il seguente comando:

```
npm install -g aws-cdk
```

Successivamente, è necessario creare una directory di progetto, quindi accedervi. Un esempio di set di comandi per creare e navigare in una directory è:

```
mkdir example-cdk-app
cd example-cdk-app
```

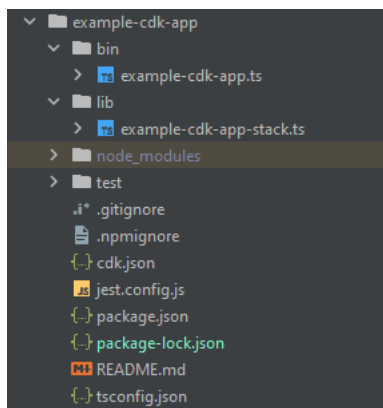
Successivamente, devi creare un'app. Il nostro servizio utilizza principalmente TypeScript. Nella directory del progetto, inserisci il seguente comando:

```
cdk init app --language typescript
```

Quando esegui questa operazione, verrà installata un'app CDK con i relativi file di inizializzazione:

```
Initializing a new git repository...
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Executing npm install...
✔ All done!
```

La struttura del progetto potrebbe essere simile a questa:



Noterai che abbiamo diverse directory importanti:

- **bin**: Il file bin iniziale creerà l'app. Non ne parleremo in questa guida.
- **lib**: La directory lib contiene i tuoi file stack. Potete pensare ai file stack come a singole unità di esecuzione. I costrutti saranno all'interno dei nostri file stack. Fondamentalmente, si tratta di risorse per un servizio che verrà attivato CloudFormation quando l'app verrà distribuita. È qui che avverrà la maggior parte della nostra codifica.

- `node_modules`: Questa directory è creata da NPM e contiene tutte le dipendenze dei pacchetti che sono state installate utilizzando il comando. `npm`

Il nostro file stack iniziale può contenere qualcosa del genere:

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
// import * as sqs from 'aws-cdk-lib/aws-sqs';

export class ExampleCdkAppStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // The code that defines your stack goes here

    // example resource
    // const queue = new sqs.Queue(this, 'ExampleCdkAppQueue', {
    //   visibilityTimeout: cdk.Duration.seconds(300)
    // });
  }
}
```

Questo è il codice standard per creare uno stack nella nostra app. La maggior parte del nostro codice in questo esempio rientrerà nell'ambito di questa classe.

Per verificare che il file stack si trovi nell'app, nella directory dell'app, esegui il seguente comando nel terminale:

```
cdk ls
```

Dovrebbe apparire un elenco dei tuoi stack. In caso contrario, potrebbe essere necessario ripetere i passaggi o consultare la documentazione ufficiale per ricevere assistenza.

Se desideri creare le modifiche al codice prima della distribuzione, puoi sempre eseguire il seguente comando nel terminale:

```
npm run build
```

Inoltre, per vedere le modifiche prima della distribuzione:

```
cdk diff
```

Prima di aggiungere il codice al file stack, eseguiremo un bootstrap. Il bootstrap ci consente di fornire risorse per il CDK prima della distribuzione dell'app. [Ulteriori informazioni su questo processo sono disponibili qui](#). Per creare un bootstrap, il comando è:

```
cdk bootstrap aws://ACCOUNT-NUMBER/REGION
```

Tip

Questo passaggio richiede diverse autorizzazioni IAM nel tuo account. Il tuo bootstrap verrà negato se non li hai. In tal caso, potrebbe essere necessario eliminare le risorse incomplete causate dal bootstrap, ad esempio il bucket S3 che genera.

Bootstrap genererà diverse risorse. Il messaggio finale sarà simile al seguente:

```
✘ Bootstrapping environment [redacted]
Trusted accounts for deployment: (none)
Trusted accounts for lookup: (none)
Using default execution policy of 'arn:aws:iam::aws:policy/AdministratorAccess'. Pass '--cloudformation-execution-policies' to customize.
CDKToolkit: creating CloudFormation changeset...
✔ Environment [redacted] bootstrapped.
```

Questa operazione viene eseguita una volta per account per regione, quindi non dovrai farlo spesso. Le risorse principali del bootstrap sono lo CloudFormation stack e il bucket Amazon S3.

Il bucket Amazon S3 viene utilizzato per archiviare file e ruoli IAM che concedono le autorizzazioni necessarie per eseguire le distribuzioni. Le risorse richieste sono definite in uno CloudFormation stack, chiamato stack bootstrap, che di solito viene denominato. CDKToolkit Come ogni CloudFormation stack, appare nella CloudFormation console una volta distribuito:

Stacks (10)				
Filter by stack name			Filter status	
			Active	☒ View nested
	Stack name	Status	Created time	Description
☐	CDKToolkit	✔ CREATE_COMPLETE	2023-07-30 21:20:19 UTC-0700	This stack includes resources needed to deploy AWS CDK apps into this environment

Lo stesso si può dire per il bucket:

	Name	AWS Region	Access	Creation date
☐	cdk-l[redacted]-assets-us-west-2	US West (Oregon) us-west-2	Bucket and objects not public	July 30, 2023, 21:20:29 (UTC-07:00)

Per importare i servizi di cui abbiamo bisogno nel nostro file stack, possiamo usare il seguente comando:

```
npm install aws-cdk-lib # V2 command
```

Tip

Se hai problemi con la V2, puoi installare le singole librerie usando i comandi V1:

```
npm install @aws-cdk/aws-appsync @aws-cdk/aws-dynamodb
```

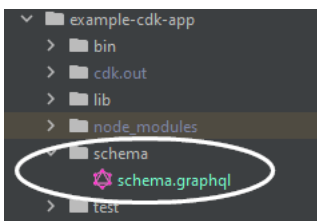
Non lo consigliamo perché la V1 è obsoleta.

Implementazione di un progetto CDK - Schema

Ora possiamo iniziare a implementare il nostro codice. Innanzitutto, dobbiamo creare il nostro schema. Puoi semplicemente creare un `.graphql` file nella tua app:

```
mkdir schema
touch schema.graphql
```

Nel nostro esempio, abbiamo incluso una directory di primo livello chiamata `schema` contenente il nostro `schema.graphql`:



All'interno del nostro schema, includiamo un semplice esempio:

```
input CreatePostInput {
  title: String
  content: String
}

type Post {
  id: ID!
  title: String
}
```



```

    content: String
  }

  type Mutation {
    createPost(input: CreatePostInput!): Post
  }

  type Query {
    getPost: [Post]
  }

```

Tornando al nostro file `stack`, dobbiamo assicurarci che siano definite le seguenti direttive di importazione:

```

import * as cdk from 'aws-cdk-lib';
import * as appsync from 'aws-cdk-lib/aws-appsync';
import * as dynamodb from 'aws-cdk-lib/aws-dynamodb';
import { Construct } from 'constructs';

```

All'interno della classe, aggiungeremo il codice per creare la nostra API GraphQL e collegarla al nostro `schema.graphql` file:

```

export class ExampleCdkAppStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // makes a GraphQL API
    const api = new appsync.GraphqlApi(this, 'post-apis', {
      name: 'api-to-process-posts',
      schema: appsync.SchemaFile.fromAsset('schema/schema.graphql'),
    });
  }
}

```

Aggiungeremo anche del codice per stampare l'URL GraphQL, la chiave API e la regione:

```

export class ExampleCdkAppStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Makes a GraphQL API construct
    const api = new appsync.GraphqlApi(this, 'post-apis', {

```

```
    name: 'api-to-process-posts',
    schema: appsync.SchemaFile.fromAsset('schema/schema.graphql'),
  });

  // Prints out URL
  new cdk.CfnOutput(this, "GraphQLAPIURL", {
    value: api.graphqlUrl
  });

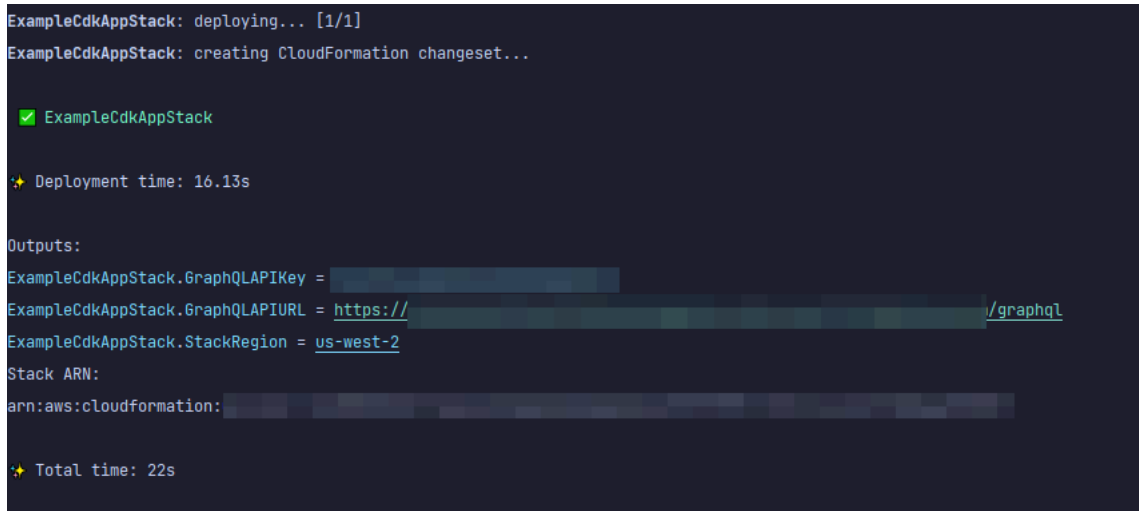
  // Prints out the AppSync GraphQL API key to the terminal
  new cdk.CfnOutput(this, "GraphQLAPIKey", {
    value: api.apiKey || ''
  });

  // Prints out the stack region to the terminal
  new cdk.CfnOutput(this, "Stack Region", {
    value: this.region
  });
}
```

A questo punto, utilizzeremo nuovamente la distribuzione della nostra app:

```
cdk deploy
```

Questo è il risultato:



```
ExampleCdkAppStack: deploying... [1/1]
ExampleCdkAppStack: creating CloudFormation changeset...

[✓] ExampleCdkAppStack

Deployment time: 16.13s

Outputs:
ExampleCdkAppStack.GraphQLAPIKey = [REDACTED]
ExampleCdkAppStack.GraphQLAPIURL = https://[REDACTED]/graphql
ExampleCdkAppStack.StackRegion = us-west-2
Stack ARN:
arn:aws:cloudformation:[REDACTED]

Total time: 22s
```

Sembra che il nostro esempio abbia avuto successo, ma controlliamo la AWS AppSync console solo per confermare:

○ api-to-process-posts	GraphQL			API_KEY

Sembra che la nostra API sia stata creata. Ora controlleremo lo schema allegato all'API:

```

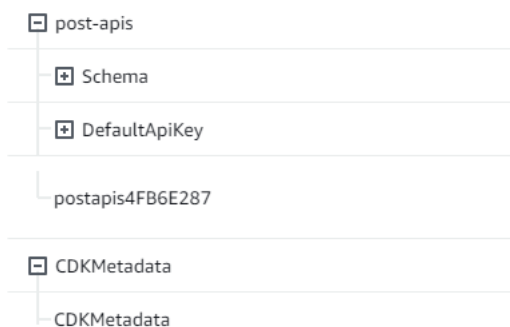
Schema
1  input CreatePostInput {
2    title: String
3    date: AWSDateTime
4  }
5
6  type Post {
7    id: ID!
8    title: String
9    date: AWSDateTime
10 }
11
12 type Mutation {
13   createPost(input: CreatePostInput!): Post
14 }
15
16 type Query {
17   getPost: [Post]
18 }

```

Questo sembra corrispondere al nostro codice dello schema, quindi ha avuto successo. Un altro modo per confermarlo dal punto di vista dei metadati è guardare lo CloudFormation stack:

○ ExampleCdkAppStack	🟢 UPDATE_COMPLETE	2023-07-30 22:13:31 UTC-0700	-
○ CDKToolkit	🟢 CREATE_COMPLETE	2023-07-30 21:20:19 UTC-0700	This stack includes resources needed to deploy AWS CDK apps into this environment

Quando distribuiamo la nostra app CDK, questa attiva risorse come CloudFormation il bootstrap. Ogni stack all'interno della nostra app viene mappato 1:1 con uno stack. CloudFormation Se torni al codice dello stack, il nome dello stack è stato preso dal nome della classe. ExampleCdkAppStack Puoi vedere le risorse che ha creato, che corrispondono anche alle nostre convenzioni di denominazione, nel nostro costruito API GraphQL:



Implementazione di un progetto CDK - Fonte dei dati

Successivamente, dobbiamo aggiungere la nostra fonte di dati. Il nostro esempio utilizzerà una tabella DynamoDB. All'interno della classe stack, aggiungeremo del codice per creare una nuova tabella:

```
export class ExampleCdkAppStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Makes a GraphQL API construct
    const api = new appsync.GraphqlApi(this, 'post-apis', {
      name: 'api-to-process-posts',
      schema: appsync.SchemaFile.fromAsset('schema/schema.graphql'),
    });

    //creates a DDB table
    const add_ddb_table = new dynamodb.Table(this, 'posts-table', {
      partitionKey: {
        name: 'id',
        type: dynamodb.AttributeType.STRING,
      },
    });

    // Prints out URL
    new cdk.CfnOutput(this, "GraphQLAPIURL", {
      value: api.graphqlUrl
    });

    // Prints out the AppSync GraphQL API key to the terminal
    new cdk.CfnOutput(this, "GraphQLAPIKey", {
      value: api.apiKey || ''
    });

    // Prints out the stack region to the terminal
    new cdk.CfnOutput(this, "Stack Region", {
      value: this.region
    });
  }
}
```

A questo punto, eseguiamo nuovamente l'implementazione:

```
cdk deploy
```

Dovremmo controllare la console DynamoDB per la nostra nuova tabella:

☐	ExampleCdkAppStack-poststable	✔ Active	id (S)	-	0		Provisioned (5)	Provisioned (5)	0 bytes	Standard

Il nome dello stack è corretto e il nome della tabella corrisponde al nostro codice. Se controlliamo nuovamente il nostro CloudFormation stack, ora vedremo la nuova tabella:

Logical ID
☒ post-apis
☒ posts-table
poststable6CB5A2E6
☒ CDKMetadata

Implementazione di un progetto CDK - Resolver

Questo esempio utilizzerà due resolver: uno per interrogare la tabella e uno per aggiungervi. Poiché utilizziamo resolver di pipeline, dovremo dichiarare due resolver di pipeline con una funzione ciascuno. Nella query, aggiungeremo il seguente codice:

```
export class ExampleCdkAppStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Makes a GraphQL API construct
    const api = new appsync.GraphqlApi(this, 'post-apis', {
      name: 'api-to-process-posts',
      schema: appsync.SchemaFile.fromAsset('schema/schema.graphql'),
    });

    //creates a DDB table
    const add_ddb_table = new dynamodb.Table(this, 'posts-table', {
      partitionKey: {
        name: 'id',
        type: dynamodb.AttributeType.STRING,
      },
    });

    // Creates a function for query
    const add_func = new appsync.AppsyncFunction(this, 'func-get-post', {
```

```

    name: 'get_posts_func_1',
    api,
    dataSource: api.addDynamoDbDataSource('table-for-posts', add_ddb_table),
    code: appsync.Code.fromInline(`
      export function request(ctx) {
        return { operation: 'Scan' };
      }

      export function response(ctx) {
        return ctx.result.items;
      }
    `),
    runtime: appsync.FunctionRuntime.JS_1_0_0,
  });

// Creates a function for mutation
const add_func_2 = new appsync.AppsyncFunction(this, 'func-add-post', {
  name: 'add_posts_func_1',
  api,
  dataSource: api.addDynamoDbDataSource('table-for-posts-2', add_ddb_table),
  code: appsync.Code.fromInline(`
    export function request(ctx) {
      return {
        operation: 'PutItem',
        key: util.dynamodb.toMapValues({id: util.autoId()}),
        attributeValues: util.dynamodb.toMapValues(ctx.args.input),
      };
    }

    export function response(ctx) {
      return ctx.result;
    }
  `),
  runtime: appsync.FunctionRuntime.JS_1_0_0,
});

// Adds a pipeline resolver with the get function
new appsync.Resolver(this, 'pipeline-resolver-get-posts', {
  api,
  typeName: 'Query',
  fieldName: 'getPost',
  code: appsync.Code.fromInline(`
    export function request(ctx) {
      return {};
    }
  `),
  runtime: appsync.FunctionRuntime.JS_1_0_0,
});

```

```

    }

    export function response(ctx) {
      return ctx.prev.result;
    }
  },
  runtime: appsync.FunctionRuntime.JS_1_0_0,
  pipelineConfig: [add_func],
});

// Adds a pipeline resolver with the create function
new appsync.Resolver(this, 'pipeline-resolver-create-posts', {
  api,
  typeName: 'Mutation',
  fieldName: 'createPost',
  code: appsync.Code.fromInline(`
    export function request(ctx) {
      return {};
    }

    export function response(ctx) {
      return ctx.prev.result;
    }
  `),
  runtime: appsync.FunctionRuntime.JS_1_0_0,
  pipelineConfig: [add_func_2],
});

// Prints out URL
new cdk.CfnOutput(this, "GraphQLAPIURL", {
  value: api.graphqlUrl
});

// Prints out the AppSync GraphQL API key to the terminal
new cdk.CfnOutput(this, "GraphQLAPIKey", {
  value: api.apiKey || ''
});

// Prints out the stack region to the terminal
new cdk.CfnOutput(this, "Stack Region", {
  value: this.region
});
}

```

```
}
```

In questo frammento, abbiamo aggiunto un risolutore di pipeline chiamato `pipeline-resolver-create-posts` con una funzione chiamata `attached`. `func-add-post` Questo è il codice che verrà aggiunto alla tabella. `Posts` L'altro risolutore di pipeline è stato chiamato `pipeline-resolver-get-posts` con una funzione chiamata `func-get-post` che recupera i dati `Posts` aggiunti alla tabella.

Lo implementeremo per aggiungerlo al servizio: AWS AppSync

```
cdk deploy
```

Controlliamo la AWS AppSync console per vedere se erano collegate alla nostra API GraphQL:

Mutation	
Field	Resolver
<code>createPost(...): Post</code>	 Pipeline
Query	
Field	Resolver
<code>getPost: [Post]</code>	 Pipeline

Sembra essere corretta. Nel codice, entrambi questi resolver erano collegati all'API GraphQL che abbiamo creato (indicata dal valore `api` props presente sia nei resolver che nelle funzioni). Nell'API GraphQL, i campi a cui abbiamo collegato i nostri resolver sono stati specificati anche nelle proprietà (definite da `typename` e `fieldName` props in ogni resolver).

Vediamo se il contenuto dei resolver è corretto a partire da: `pipeline-resolver-get-posts`

▼ Resolver code

1

2 ▼

3

4

5

6 ▼

7

8

9

```
export function request(ctx) {  
  return {};  
}  
  
export function response(ctx) {  
  return ctx.prev.result;  
}
```

APPSYNC_JS Ln 1, Col 1 ⓧ Errors: 0 ⚠ Warnings: 0

Functions

Each function is executed in sequence and can execute a single operation against a data source.

🔍 Find by function name

add_posts_func_1 Edit

Description

-

► Function code read-only

I gestori prima e dopo corrispondono al nostro valore props. code Possiamo anche vedere una funzione chiamata `add_posts_func_1`, che corrisponde al nome della funzione che abbiamo collegato nel resolver.

Diamo un'occhiata al contenuto del codice di quella funzione:

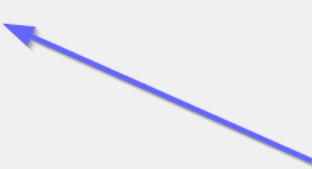
add_posts_func_1 Edit

Description

-

▼ **Function code** read-only

```
1
2   export function request(ctx) {
3     return {
4       operation: 'PutItem',
5       key: util.dynamodb.toMapValues({id: util.autoId()}),
6       attributeValues: util.dynamodb.toMapValues(ctx.args.input),
7     };
8   }
9
10  export function response(ctx) {
11    return ctx.result;
12  }
13
```



Questo corrisponde agli code oggetti di scena della `add_posts_func_1` funzione. La nostra query è stata caricata correttamente, quindi controlliamo la query:

▼ Resolver code

```
1
2 export function request(ctx) {
3   return {};
4 }
5
6 export function response(ctx) {
7   return ctx.prev.result;
8 }
9
```

APPSYNC_JS Ln 1, Col 1  Errors: 0  Warnings: 0

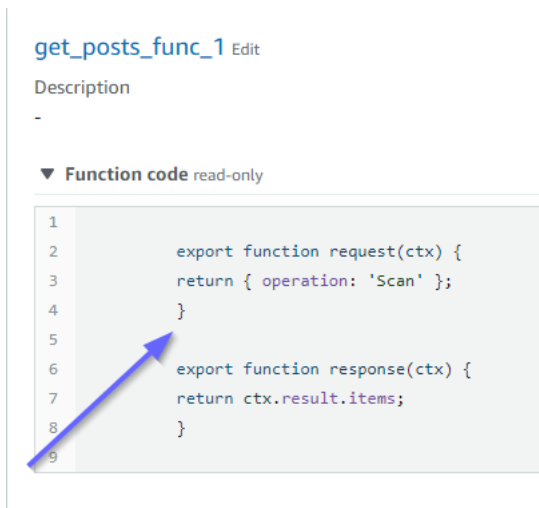
Functions
Each function is executed in sequence and can execute a single operation against a data source.

[get_posts_func_1](#) Edit

Description
-

► **Function code** read-only

Anche questi corrispondono al codice. Se osserviamo `get_posts_func_1`:



Sembra che tutto sia a posto. Per confermarlo dal punto di vista dei metadati, possiamo ricontrollare il nostro stack: CloudFormation

Logical ID
+ post-apis
+ posts-table
+ func-get-post
+ func-add-post
+ pipeline-resolver-get-posts
+ pipeline-resolver-create-posts
+ CDKMetadata

Ora, dobbiamo testare questo codice eseguendo alcune richieste.

Implementazione di un progetto CDK - Richieste

Per testare la nostra app nella AWS AppSync console, abbiamo effettuato una query e una mutazione:

```

1 query MyQuery {
2   getPost {
3     id
4     date
5     title
6   }
7 }
8
9 mutation MyMutation {
10  createPost(input: {date: "1970-01-01T12:30:00.000Z", title: "first post"}) {
11    date
12    id
13    title
14  }
15 }
16

```

MyMutation contiene un'createPost operazione con gli argomenti 1970-01-01T12:30:00.000Z e first post. Restituisce l'id e il titolo che abbiamo passato e il valore id generato automaticamente. L'esecuzione della mutazione produce il risultato:

```

{
  "data": {
    "createPost": {
      "date": "1970-01-01T12:30:00.000Z",
      "id": "4dc1c2dd-0aa3-4055-9eca-7c140062ada2",
      "title": "first post"
    }
  }
}

```

Se controlliamo rapidamente la tabella DynamoDB, possiamo vedere la nostra voce nella tabella quando la scansiamo:

☐	id (String)	date	title
☐	9f62c4dd-49d5-48d5-b835-143284c72fe0	1970-01-01T12:30:00.000Z	first post

Tornando alla AWS AppSync console, se eseguiamo la query per recuperarloPost, otteniamo il seguente risultato:

```

{
  "data": {
    "getPost": [
      {
        "id": "9f62c4dd-49d5-48d5-b835-143284c72fe0",
        "date": "1970-01-01T12:30:00.000Z",

```

```
      "title": "first post"
    }
  ]
}
}
```

Utilizzo di abbonamenti per applicazioni di dati in tempo reale in AWS AppSync

Important

A partire dal 13 marzo 2025, puoi creare un' PubSub API in tempo reale basata su Events. WebSockets AWS AppSync Per ulteriori informazioni, consulta [Pubblicare eventi tramite WebSocket](#) nella AWS AppSync Events Developer Guide.

AWS AppSync consente di utilizzare gli abbonamenti per implementare aggiornamenti in tempo reale delle applicazioni, notifiche push, ecc. Quando i client richiamano le operazioni di sottoscrizione GraphQL, viene stabilita e gestita automaticamente una connessione WebSocket sicura da AWS AppSync. Le applicazioni possono quindi distribuire i dati in tempo reale da una fonte di dati agli abbonati, gestendo al contempo AWS AppSync i requisiti di connessione e scalabilità dell'applicazione. Le seguenti sezioni ti mostreranno come funzionano gli abbonamenti. AWS AppSync

Direttive di sottoscrizione allo schema GraphQL

Le sottoscrizioni in AWS AppSync vengono richiamate come risposta a una mutazione. Ciò significa che puoi creare qualsiasi fonte di dati in tempo AWS AppSync reale specificando una direttiva dello schema GraphQL su una mutazione.

Le librerie AWS Amplify client gestiscono automaticamente la gestione delle connessioni in abbonamento. Le librerie utilizzano pure WebSockets come protocollo di rete tra il client e il servizio.

Note

Per controllare l'autorizzazione al momento della connessione a un abbonamento, puoi utilizzare AWS Identity and Access Management (IAM) AWS Lambda, i pool di identità di

Amazon Cognito o i pool di utenti Amazon Cognito per l'autorizzazione a livello di campo. Per controlli di accesso dettagliati sugli abbonamenti, puoi collegare resolver ai campi di abbonamento ed eseguire la logica utilizzando l'identità del chiamante e le fonti di dati. AWS AppSync Per ulteriori informazioni, consulta [Configurazione dell'autorizzazione e dell'autenticazione per proteggere GraphQL APIs](#).

Le sottoscrizioni vengono attivate da mutazioni e il set di selezioni delle mutazioni viene inviato ai sottoscrittori.

L'esempio seguente mostra come usare sottoscrizioni GraphQL. Non specifica un'origine dati perché l'origine dati potrebbe essere Lambda, Amazon DynamoDB o Amazon Service. OpenSearch

Per iniziare con gli abbonamenti, devi aggiungere un punto di ingresso dell'abbonamento allo schema come segue:

```
schema {  
  query: Query  
  mutation: Mutation  
  subscription: Subscription  
}
```

Supponiamo di avere un sito di post di blog e di voler sottoscrivere nuovi blog e le modifiche a blog esistenti. A questo scopo, aggiungere la definizione Subscription seguente allo schema:

```
type Subscription {  
  addedPost: Post  
  updatedPost: Post  
  deletedPost: Post  
}
```

Supponiamo inoltre di avere le mutazioni seguenti:

```
type Mutation {  
  addPost(id: ID! author: String! title: String content: String url: String): Post!  
  updatePost(id: ID! author: String! title: String content: String url: String ups:  
    Int! downs: Int! expectedVersion: Int!): Post!  
  deletePost(id: ID!): Post!  
}
```

Puoi impostare questi campi come elementi in tempo reale aggiungendo una direttiva `@aws_subscribe(mutations: ["mutation_field_1", "mutation_field_2"])` per ognuna delle sottoscrizioni per cui vuoi ricevere notifiche, in questo modo:

```
type Subscription {  
  addedPost: Post  
    @aws_subscribe(mutations: ["addPost"])  
  updatedPost: Post  
    @aws_subscribe(mutations: ["updatePost"])  
  deletedPost: Post  
    @aws_subscribe(mutations: ["deletePost"])  
}
```

Poiché `@aws_subscribe(mutations: ["", .., ""])` richiede una serie di input di mutazione, è possibile specificare più mutazioni, che avviano una sottoscrizione. Se esegui la sottoscrizione da un client, la query GraphQL può essere simile alla seguente:

```
subscription NewPostSub {  
  addedPost {  
    __typename  
    version  
    title  
    content  
    author  
    url  
  }  
}
```

Questa richiesta di sottoscrizione è necessaria per le connessioni e gli strumenti dei client.

Con il WebSockets client puro, il filtraggio del set di selezione viene eseguito per client, poiché ogni client può definire il proprio set di selezione. In questo caso, il set di selezione della sottoscrizione deve essere un sottoinsieme del set di selezione delle mutazioni. Ad esempio, un abbonamento `addedPost{author title}` collegato alla mutazione `addPost(...){id author title url version}` riceve solo l'autore e il titolo del post. Non riceve gli altri campi. Tuttavia, se per la mutazione manca l'autore nel set di selezione, il sottoscrittore ottiene un valore `null` per il campo autore (o un errore nel caso in cui il campo autore sia definito come richiesto/non-null nello schema).

Il set di selezione dell'abbonamento è essenziale quando si utilizza pure WebSockets. Se un campo non è definito in modo esplicito nell'abbonamento, AWS AppSync non restituisce il campo.

Nell'esempio precedente, le sottoscrizioni non dispongono di argomenti. Supponiamo che lo schema sia simile al seguente:

```
type Subscription {  
  updatedPost(id:ID! author:String): Post  
  @aws_subscribe(mutations: ["updatePost"])  
}
```

In questo caso, il client definisce una sottoscrizione come segue:

```
subscription UpdatedPostSub {  
  updatedPost(id:"XYZ", author:"ABC") {  
    title  
    content  
  }  
}
```

Il tipo restituito di un campo `subscription` nello schema deve corrispondere al tipo restituito del campo della mutazione corrispondente. Questo è stato mostrato nell'esempio precedente, in quanto sia `addPost` sia `addedPost` hanno restituito `Post` come tipo.

Per configurare gli abbonamenti sul client, consulta. [Creazione di un'applicazione client utilizzando il client Amplify](#)

Utilizzo degli argomenti di sottoscrizione

Una parte importante dell'utilizzo degli abbonamenti GraphQL è capire quando e come utilizzare gli argomenti. È possibile apportare lievi modifiche per modificare come e quando notificare ai client le mutazioni che si sono verificate. A tale scopo, consultate lo schema di esempio tratto dal capitolo quickstart, che crea «Todos». Per questo schema di esempio, vengono definite le seguenti mutazioni:

```
type Mutation {  
  createTodo(input: CreateTodoInput!): Todo  
  updateTodo(input: UpdateTodoInput!): Todo  
  deleteTodo(input: DeleteTodoInput!): Todo  
}
```

Nell'esempio predefinito, i client possono sottoscrivere gli aggiornamenti di any Todo utilizzando il comando `onUpdateTodo subscription` senza argomenti:

```
subscription OnUpdateTodo {  
  onUpdateTodo {  
    description  
    id  
    name  
    when  
  }  
}
```

Puoi filtrare i tuoi subscription utilizzando i relativi argomenti. Ad esempio, per attivare a solo subscription quando ID viene aggiornato un todo con uno specifico, specifica il ID valore:

```
subscription OnUpdateTodo {  
  onUpdateTodo(id: "a-todo-id") {  
    description  
    id  
    name  
    when  
  }  
}
```

Puoi anche passare più argomenti. Ad esempio, quanto segue subscription dimostra come ricevere notifiche di eventuali Todo aggiornamenti in un luogo e in un'ora specifici:

```
subscription todosAtHome {  
  onUpdateTodo(when: "tomorrow", where: "at home") {  
    description  
    id  
    name  
    when  
    where  
  }  
}
```

Nota che tutti gli argomenti sono opzionali. Se non specifichi alcun argomento nel tuosubscription, sarai iscritto a tutti Todo gli aggiornamenti che avvengono nella tua applicazione. Tuttavia, puoi aggiornare la definizione subscription del campo per richiedere l'IDargomento. Ciò forzerebbe la risposta di uno specifico todo anziché di tutti i todo s:

```
onUpdateTodo(  
  id: ID!,
```

```
    name: String,  
    when: String,  
    where: String,  
    description: String  
  ): Todo
```

Il valore null dell'argomento ha un significato

Quando si effettua una query di sottoscrizione AWS AppSync, un valore di `null` argomento filtrerà i risultati in modo diverso rispetto all'omissione completa dell'argomento.

Torniamo all'esempio di API todos in cui potremmo creare todos. Vedi lo schema di esempio tratto dal capitolo quickstart.

Modifichiamo il nostro schema per includere un nuovo `owner` campo, sul `Todo` tipo, che descriva chi è il proprietario. Il `owner` campo non è obbligatorio e può essere solo impostato `UpdateTodoInput`. Vedi la seguente versione semplificata dello schema:

```
type Todo {  
  id: ID!  
  name: String!  
  when: String!  
  where: String!  
  description: String!  
  owner: String  
}  
  
input CreateTodoInput {  
  name: String!  
  when: String!  
  where: String!  
  description: String!  
}  
  
input UpdateTodoInput {  
  id: ID!  
  name: String  
  when: String  
  where: String  
  description: String  
  owner: String  
}
```

```
type Subscription {
  onUpdateTodo(
    id: ID!,
    name: String!,
    when: String!,
    where: String!,
    description: String!
  ): Todo @aws_subscribe(mutations: ["updateTodo"])
}
```

Il seguente abbonamento restituisce tutti gli Todo aggiornamenti:

```
subscription MySubscription {
  onUpdateTodo {
    description
    id
    name
    when
    where
  }
}
```

Se modifichi la sottoscrizione precedente per aggiungere l'argomento del campo `owner: null`, ora stai facendo una domanda diversa. Questo abbonamento ora consente al cliente di ricevere una notifica di tutti gli Todo aggiornamenti per i quali non è stato fornito alcun proprietario.

```
subscription MySubscription {
  onUpdateTodo(owner: null) {
    description
    id
    name
    when
    where
  }
}
```

Note

A partire dal 1° gennaio 2022, MQTT over non WebSockets è più disponibile come protocollo per gli abbonamenti GraphQL in. AWS AppSync APIs Pure WebSockets è l'unico protocollo supportato in. AWS AppSync

I client basati sull' AWS AppSync SDK o sulle librerie Amplify, rilasciati dopo novembre 2019, utilizzano automaticamente pure per impostazione predefinita. WebSockets L'aggiornamento dei client alla versione più recente consente loro di utilizzare il motore puro. AWS AppSync WebSockets

Pure offre una dimensione del payload più grande (240 KB), una più ampia varietà di opzioni client e metriche WebSockets migliorate. CloudWatch Per ulteriori informazioni sull'utilizzo dei WebSocket client pure, consulta. [WebSocket Creazione di un client in tempo reale in AWS AppSync](#)

Creazione di sistemi generici con pub/sub APIs tecnologia serverless in WebSockets AWS AppSync

Important

A partire dal 13 marzo 2025, puoi creare un' PubSub API in tempo reale basata su WebSockets AWS AppSync Events. Per ulteriori informazioni, consulta [Pubblicare eventi tramite WebSocket](#) nella AWS AppSync Events Developer Guide.

Alcune applicazioni richiedono solo che WebSocket APIs i clienti ascoltino un canale o un argomento specifico. I dati JSON generici senza una forma specifica o requisiti fortemente tipizzati possono essere inviati ai clienti che ascoltano uno di questi canali secondo uno schema di pubblicazione (pub/sub) puro e semplice.

Utilizzalo AWS AppSync per implementare soluzioni semplici pub/sub WebSocket APIs con poca o nessuna conoscenza di GraphQL in pochi minuti generando automaticamente codice GraphQL sia sul backend dell'API che sul lato client.

Crea e configura pub-sub APIs

Per iniziare, procedi come segue:

1. Accedi a Console di gestione AWS e apri la [AppSync console](#).
 - Nel pannello di controllo, scegliere Create API (Crea API).
2. Nella schermata successiva, scegli Crea un'API in tempo reale, quindi scegli Avanti.
3. Inserisci un nome descrittivo per la tua pub/sub API.

4. Puoi abilitare le funzionalità [API private](#), ma per ora ti consigliamo di tenerla disattivata. Scegli Next (Successivo).
5. Puoi scegliere di generare automaticamente un' pub/sub API funzionante utilizzando WebSockets. Ti consigliamo di tenere disattivata questa funzionalità anche per ora. Scegli Next (Successivo).
6. Scegli Crea API e attendi un paio di minuti. Nel tuo account verrà creata una nuova API AWS AppSync pub/sub preconfigurata. AWS

L'API utilizza i AWS AppSync resolver locali integrati (per ulteriori informazioni sull'utilizzo dei [resolver locali](#), consulta [Tutorial: Local Resolvers nella AWS AppSync Developer Guide](#)) per gestire più pub/sub canali e WebSocket connessioni temporanee, che distribuiscono e filtrano automaticamente i dati ai client abbonati solo in base al nome del canale. Le chiamate API sono autorizzate con una chiave API.

Dopo l'implementazione dell'API, ti vengono presentati un paio di passaggi aggiuntivi per generare il codice client e integrarlo con l'applicazione client. Per un esempio su come integrare rapidamente un client, questa guida utilizzerà una semplice applicazione web React.

1. Inizia creando un'app React standard usando [NPM](#) sul tuo computer locale:

```
$ npx create-react-app mypubsub-app  
$ cd mypubsub-app
```

Note

Questo esempio utilizza le librerie [Amplify](#) per connettere i client all'API di backend. Tuttavia non è necessario creare un progetto Amplify CLI localmente. Sebbene React sia il client preferito in questo esempio, le librerie Amplify supportano anche i client iOS, Android e Flutter, fornendo le stesse funzionalità in questi diversi runtime. [I client Amplify supportati forniscono semplici astrazioni per interagire con i backend dell'API AWS AppSync GraphQL con poche righe di codice, incluse WebSocket funzionalità integrate completamente compatibili con il protocollo in tempo reale:AWS AppSync WebSocket](#)

```
$ npm install @aws-amplify/api
```

2. Nella AWS AppSync console, seleziona JavaScript, quindi Scarica per scaricare un singolo file con i dettagli di configurazione dell'API e il codice operativo GraphQL generato.

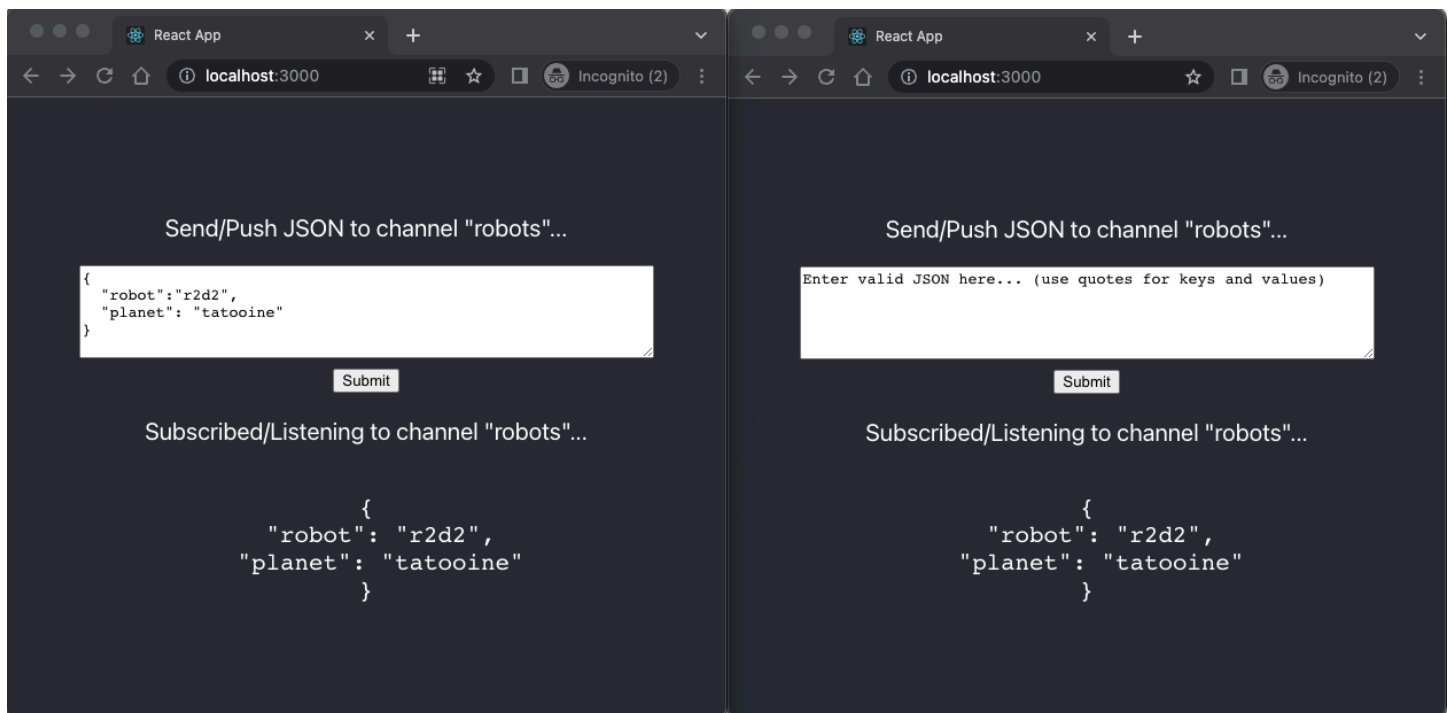
3. Copia il file scaricato `/src` nella cartella del tuo progetto React.
4. Quindi, sostituisci il contenuto del `src/App.js` file boilerplate esistente con il codice client di esempio disponibile nella console.
5. Utilizzate il seguente comando per avviare l'applicazione localmente:

```
$ npm start
```

6. Per testare l'invio e la ricezione di dati in tempo reale, apri due finestre del browser e accedi **`localhost:3000`**. L'applicazione di esempio è configurata per inviare dati JSON generici a un canale codificato denominato. **`robots`**
7. In una delle finestre del browser, inserisci il seguente blob JSON nella casella di testo, quindi fai clic su Invia:

```
{
  "robot": "r2d2",
  "planet": "tatooine"
}
```

Entrambe le istanze del browser sono iscritte al **`robots`** canale e ricevono i dati pubblicati in tempo reale, visualizzati nella parte inferiore dell'applicazione web:



Tutto il codice API GraphQL necessario, inclusi lo schema, i resolver e le operazioni, viene generato automaticamente per consentire un caso d'uso generico. pub/sub Sul backend, i dati vengono pubblicati sull'endpoint in tempo reale AWS AppSync dell'utente con una mutazione GraphQL come la seguente:

```
mutation PublishData {  
  publish(data: "{\"msg\": \"hello world!\"}", name: "channel") {  
    data  
    name  
  }  
}
```

Gli abbonati accedono ai dati pubblicati inviati al canale temporaneo specifico con un abbonamento GraphQL correlato:

```
subscription SubscribeToData {  
  subscribe(name:"channel") {  
    name  
    data  
  }  
}
```

Implementazione di pub-sub APIs nelle applicazioni esistenti

Nel caso in cui sia sufficiente implementare una funzionalità in tempo reale in un'applicazione esistente, questa configurazione pub/sub API generica può essere facilmente integrata in qualsiasi applicazione o tecnologia API. Sebbene vi siano vantaggi nell'utilizzare un singolo endpoint API per accedere, manipolare e combinare in modo sicuro i dati provenienti da una o più fonti di dati in una singola chiamata di rete con GraphQL, non è necessario convertire o ricostruire da zero un'applicazione esistente basata su REST per sfruttare le funzionalità in tempo reale di GraphQL. AWS AppSync Ad esempio, potresti avere un carico di lavoro CRUD esistente in un endpoint API separato con i client che inviano e ricevono messaggi o eventi dall'applicazione esistente all' pub/sub API generica solo in tempo reale e per pub/sub scopi.

Definizione di filtri di sottoscrizione avanzati in AWS AppSync

⚠ Important

A partire dal 13 marzo 2025, puoi creare un' PubSub API in tempo reale basata su WebSockets AWS AppSync Events. Per ulteriori informazioni, consulta [Pubblicare eventi tramite WebSocket](#) nella AWS AppSync Events Developer Guide.

Inoltre AWS AppSync, puoi definire e abilitare la logica aziendale per il filtraggio dei dati sul backend direttamente nei resolver di sottoscrizione dell'API GraphQL utilizzando filtri che supportano operatori logici aggiuntivi. È possibile configurare questi filtri, a differenza degli argomenti di sottoscrizione definiti nella query di sottoscrizione nel client. Per ulteriori informazioni sull'utilizzo degli argomenti di sottoscrizione, vedere [Utilizzo degli argomenti di sottoscrizione](#). Per un elenco degli operatori, vedere [AWS AppSync riferimento all'utilità del modello di mappatura del resolver](#).

Ai fini di questo documento, suddividiamo il filtraggio dei dati in tempo reale nelle seguenti categorie:

- Filtraggio di base: filtraggio basato su argomenti definiti dal client nella query di sottoscrizione.
- Filtraggio avanzato: filtraggio basato sulla logica definita centralmente nel backend del servizio. AWS AppSync

Le sezioni seguenti spiegano come configurare i filtri di abbonamento avanzati e ne mostrano l'uso pratico.

Definizione degli abbonamenti nello schema GraphQL

Per utilizzare i filtri di sottoscrizione avanzati, è necessario definire l'abbonamento nello schema GraphQL, quindi definire il filtro avanzato utilizzando un'estensione di filtro. Per illustrare come funziona il filtraggio avanzato degli abbonamenti AWS AppSync, usa il seguente schema GraphQL, che definisce un'API del sistema di gestione dei ticket, come esempio:

```
type Ticket {  
  id: ID  
  createdAt: AWSDateTime  
  content: String  
  severity: Int  
  priority: Priority  
  category: String
```

```
  group: String
  status: String
}

type Mutation {
  createTicket(input: TicketInput): Ticket
}

type Query {
  getTicket(id: ID!): Ticket
}

type Subscription {
  onSpecialTicketCreated: Ticket @aws_subscribe(mutations: ["createTicket"])
  onGroupTicketCreated(group: String!): Ticket @aws_subscribe(mutations:
    ["createTicket"])
}

enum Priority {
  none
  lowest
  low
  medium
  high
  highest
}

input TicketInput {
  content: String
  severity: Int
  priority: Priority
  category: String
  group: String
}
```

Supponete di creare una fonte di NONE dati per la vostra API, quindi di collegare un resolver alla mutazione utilizzando questa fonte di dati. `createTicket` I tuoi gestori potrebbero avere questo aspetto:

```
import { util } from '@aws-appsync/utils';
```

```
export function request(ctx) {
  return {
    payload: {
      id: util.autoId(),
      createdAt: util.time.nowISO8601(),
      status: 'pending',
      ...ctx.args.input,
    },
  };
}

export function response(ctx) {
  return ctx.result;
}
```

Note

I filtri avanzati sono abilitati nel gestore del resolver GraphQL in un determinato abbonamento. [Per ulteriori informazioni, consulta Resolver reference.](#)

Per implementare il comportamento del filtro avanzato, è necessario utilizzare la `extensions.setSubscriptionFilter()` funzione per definire un'espressione di filtro valutata rispetto ai dati pubblicati da una mutazione GraphQL che potrebbe interessare i client sottoscritti. [Per ulteriori informazioni sulle estensioni di filtraggio, consulta Estensioni.](#)

La sezione seguente spiega come utilizzare le estensioni di filtraggio per implementare filtri avanzati.

Creazione di filtri di abbonamento avanzati utilizzando estensioni di filtraggio

I filtri avanzati sono scritti in JSON nel gestore delle risposte dei resolver dell'abbonamento. I filtri possono essere raggruppati in un elenco chiamato `filterGroup`. I filtri vengono definiti utilizzando almeno una regola, ciascuna con campi, operatori e valori. Definiamo un nuovo resolver `onSpecialTicketCreated` che imposta un filtro avanzato. È possibile configurare più regole in un filtro che vengono valutate utilizzando la logica AND, mentre più filtri in un gruppo di filtri vengono valutati utilizzando la logica OR:

```
import { util, extensions } from '@aws-appsync/utils';
```

```

export function request(ctx) {
  // simplify return null for the payload
  return { payload: null };
}

export function response(ctx) {
  const filter = {
    or: [
      { severity: { ge: 7 }, priority: { in: ['high', 'medium'] } },
      { category: { eq: 'security' }, group: { in: ['admin', 'operators'] } },
    ],
  };
  extensions.setSubscriptionFilter(util.transform.toSubscriptionFilter(filter));

  // important: return null in the response
  return null;
}

```

In base ai filtri definiti nell'esempio precedente, i ticket importanti vengono automaticamente inviati ai client API sottoscritti se viene creato un ticket con:

- **priority** livello o high medium

AND

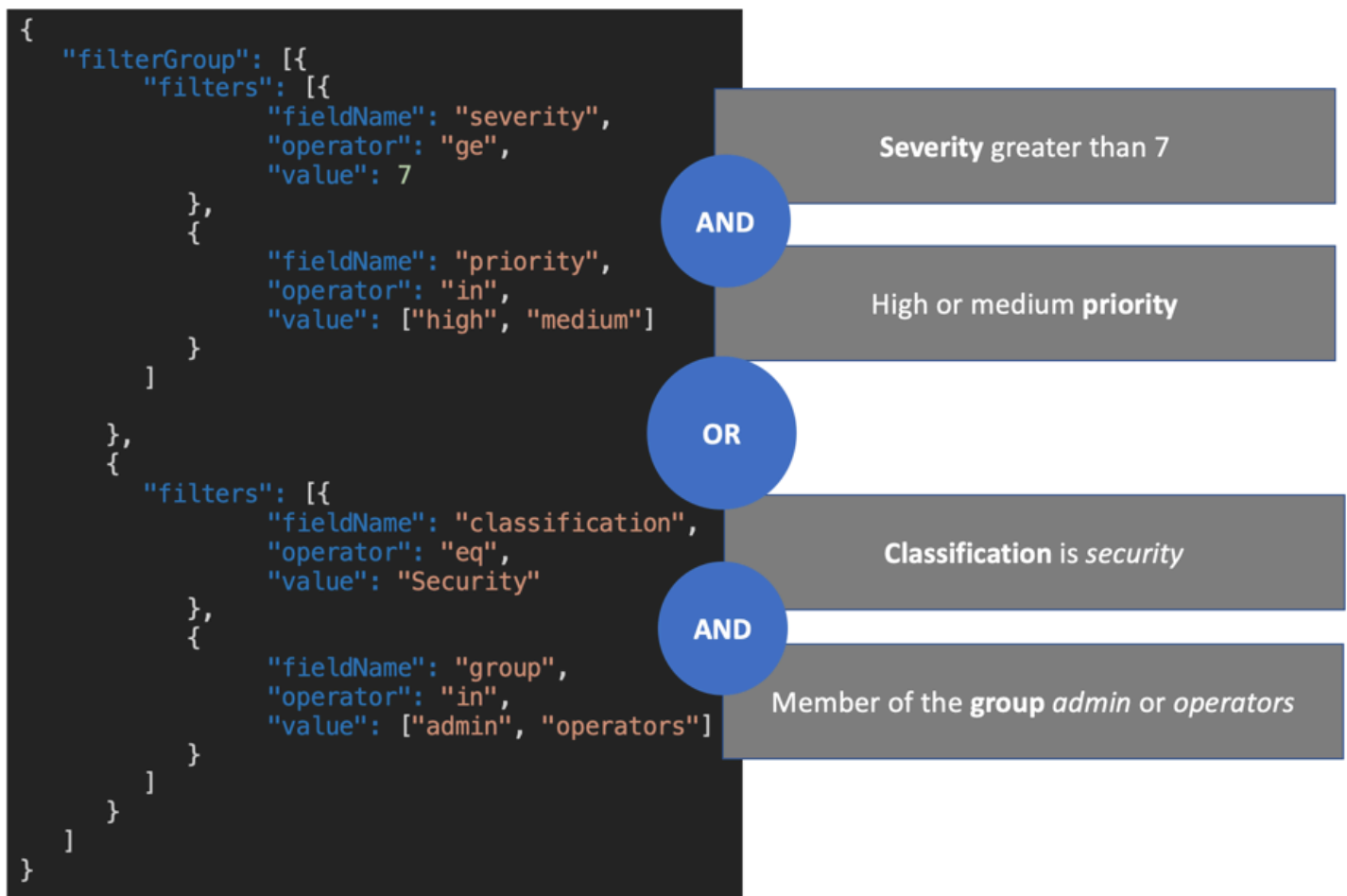
- **severity** livello maggiore o uguale a 7 (ge)

O

- **classification** biglietto impostato su Security

AND

- **group** assegnazione impostata su o admin operators



I filtri definiti nel resolver di sottoscrizione (filtro avanzato) hanno la precedenza sui filtri basati solo sugli argomenti di sottoscrizione (filtro di base). [Per ulteriori informazioni sull'utilizzo degli argomenti di sottoscrizione, vedere Utilizzo degli argomenti di sottoscrizione](#).

Se un argomento è definito e richiesto nello schema GraphQL dell'abbonamento, il filtraggio basato sull'argomento specificato avviene solo se l'argomento è definito come regola nel metodo del resolver. `extensions.setSubscriptionFilter()` Tuttavia, se non sono presenti metodi di `extensions` filtraggio nel resolver di sottoscrizione, gli argomenti definiti nel client vengono utilizzati solo per il filtraggio di base. Non è possibile utilizzare contemporaneamente il filtro di base e il filtro avanzato.

Puoi utilizzare la [contextvariabile](#) nella logica di estensione del filtro dell'abbonamento per accedere alle informazioni contestuali sulla richiesta. Ad esempio, quando utilizzi gli autorizzatori personalizzati Amazon Cognito User Pools, OIDC o Lambda per l'autorizzazione, puoi recuperare informazioni sui tuoi utenti nel momento in cui viene stabilito l'abbonamento. `context.identity` Puoi utilizzare queste informazioni per stabilire filtri in base all'identità degli utenti.

Supponiamo ora di voler implementare il comportamento di filtro avanzato `peronGroupTicketCreated`. L'`onGroupTicketCreated` abbonamento richiede un `group` nome obbligatorio come argomento. Una volta creati, ai ticket viene assegnato automaticamente uno `pending` stato. Puoi impostare un filtro di abbonamento per ricevere solo i biglietti appena creati che appartengono al gruppo fornito:

```
import { util, extensions } from '@aws-appsync/utils';

export function request(ctx) {
  // simplify return null for the payload
  return { payload: null };
}

export function response(ctx) {
  const filter = { group: { eq: ctx.args.group }, status: { eq: 'pending' } };
  extensions.setSubscriptionFilter(util.transform.toSubscriptionFilter(filter));

  return null;
}
```

Quando i dati vengono pubblicati utilizzando una mutazione come nell'esempio seguente:

```
mutation CreateTicket {
  createTicket(input: {priority: medium, severity: 2, group: "aws"}) {
    id
    priority
    severity
    status
    group
    createdAt
  }
}
```

I clienti abbonati attendono che i dati vengano trasmessi automaticamente non WebSockets appena viene creato un ticket con la mutazione: `createTicket`

```
subscription OnGroup {
  onGroupTicketCreated(group: "aws") {
    category
    status
    severity
  }
}
```

```
    priority
    id
    group
    createdAt
    content
  }
}
```

I client possono essere sottoscritti senza argomenti perché la logica di filtraggio è implementata nel AWS AppSync servizio con un filtraggio avanzato, che semplifica il codice client. I client ricevono i dati solo se vengono soddisfatti i criteri di filtro definiti.

Definizione di filtri avanzati per i campi dello schema annidati

È possibile utilizzare il filtro di sottoscrizione avanzato per filtrare i campi dello schema annidati. Supponiamo di aver modificato lo schema della sezione precedente per includere i tipi di posizione e indirizzo:

```
type Ticket {
  id: ID
  createdAt: AWSDateTime
  content: String
  severity: Int
  priority: Priority
  category: String
  group: String
  status: String
  location: ProblemLocation
}

type Mutation {
  createTicket(input: TicketInput): Ticket
}

type Query {
  getTicket(id: ID!): Ticket
}

type Subscription {
  onSpecialTicketCreated: Ticket @aws_subscribe(mutations: ["createTicket"])
  onGroupTicketCreated(group: String!): Ticket @aws_subscribe(mutations:
    ["createTicket"])
}
```

```

type ProblemLocation {
  address: Address
}

type Address {
  country: String
}

enum Priority {
  none
  lowest
  low
  medium
  high
  highest
}

input TicketInput {
  content: String
  severity: Int
  priority: Priority
  category: String
  group: String
  location: AWSJSON
}

```

Con questo schema, è possibile utilizzare un `.` separatore per rappresentare la nidificazione.

L'esempio seguente aggiunge una regola di filtro per un campo dello schema annidato in.

`location.address.country` L'abbonamento verrà attivato se l'indirizzo del ticket è impostato su: USA

```

import { util, extensions } from '@aws-appsync/utils';

export const request = (ctx) => ({ payload: null });

export function response(ctx) {
  const filter = {
    or: [
      { severity: { ge: 7 }, priority: { in: ['high', 'medium'] } },
      { category: { eq: 'security' }, group: { in: ['admin', 'operators'] } },
      { 'location.address.country': { eq: 'USA' } },
    ],
  };
}

```



```
extensions.setSubscriptionFilter(util.transform.toSubscriptionFilter(filter));
return null;
}
```

Nell'esempio precedente, `location` rappresenta il livello di nidificazione uno, `address` rappresenta il livello di nidificazione due e `country` rappresenta il livello di nidificazione tre, tutti separati dal separatore. .

Puoi testare questo abbonamento usando la mutazione: `createTicket`

```
mutation CreateTicketInUSA {
  createTicket(input: {location: "{\"address\":{\"country\":\"USA\"}}"}) {
    category
    content
    createdAt
    group
    id
    location {
      address {
        country
      }
    }
    priority
    severity
    status
  }
}
```

Definizione di filtri avanzati dal client

È possibile utilizzare il filtraggio di base in GraphQL [con](#) argomenti di sottoscrizione. Il client che effettua la chiamata nella query di sottoscrizione definisce i valori degli argomenti. Quando i filtri avanzati sono abilitati in un resolver di AWS AppSync sottoscrizione con il `extensions` filtro, i filtri di backend definiti nel resolver hanno la precedenza e la priorità.

Configura filtri avanzati dinamici e definiti dal client utilizzando un argomento nella sottoscrizione. `filter` Quando configuri questi filtri, devi aggiornare lo schema GraphQL in modo che rifletta il nuovo argomento:

```
...
type Subscription {
```

```

    onSpecialTicketCreated(filter: String): Ticket
      @aws_subscribe(mutations: ["createTicket"])
    }
    ...

```

Il client può quindi inviare una richiesta di sottoscrizione come nell'esempio seguente:

```

subscription onSpecialTicketCreated($filter: String) {
  onSpecialTicketCreated(filter: $filter) {
    id
    group
    description
    priority
    severity
  }
}

```

È possibile configurare la variabile di query come nell'esempio seguente:

```

{"filter" : "{\"severity\":{\"le\":2}}"}

```

L'utilità `util.transform.toSubscriptionFilter()` resolver può essere implementata nel modello di mappatura delle risposte all'abbonamento per applicare il filtro definito nell'argomento dell'abbonamento per ogni client:

```

import { util, extensions } from '@aws-appsync/utils';

export function request(ctx) {
  // simply return null for the payload
  return { payload: null };
}

export function response(ctx) {
  const filter = ctx.args.filter;
  extensions.setSubscriptionFilter(util.transform.toSubscriptionFilter(filter));
  return null;
}

```

Con questa strategia, i clienti possono definire i propri filtri che utilizzano una logica di filtraggio avanzata e operatori aggiuntivi. I filtri vengono assegnati quando un determinato client richiama la

query di sottoscrizione in una connessione sicura. WebSocket [Per ulteriori informazioni sull'utilità di trasformazione per un filtraggio avanzato, incluso il formato del payload della variabile di `filter` query, consulta la panoramica sui resolver.](#) JavaScript

Restrizioni di filtraggio avanzate aggiuntive

Di seguito sono riportati diversi casi d'uso in cui vengono applicate restrizioni aggiuntive ai filtri avanzati:

- I filtri avanzati non supportano il filtraggio per gli elenchi di oggetti di primo livello. In questo caso d'uso, i dati pubblicati relativi alla mutazione verranno ignorati per gli abbonamenti avanzati.
- AWS AppSync supporta fino a cinque livelli di nidificazione. I filtri sui campi dello schema che hanno superato il quinto livello di nidificazione verranno ignorati. Prendi la risposta GraphQL qui sotto. Il `continent` field in ingresso `venue.address.country.metadata.continent` è consentito perché si tratta di un nido di livello cinque. Tuttavia, `financial` `venue.address.country.metadata.capital.financial` è un nido di livello sei, quindi il filtro non funzionerà:

```
{
  "data": {
    "onCreateFilterEvent": {
      "venue": {
        "address": {
          "country": {
            "metadata": {
              "capital": {
                "financial": "New York"
              },
              "continent" : "North America"
            }
          },
          "state": "WA"
        },
        "builtYear": 2023
      },
      "private": false,
    }
  }
}
```

Annullamento dell'iscrizione alle WebSocket connessioni utilizzando i filtri in AWS AppSync

Important

A partire dal 13 marzo 2025, puoi creare un' PubSub API in tempo reale basata su WebSockets AWS AppSync Events. Per ulteriori informazioni, consulta [Pubblicare eventi tramite WebSocket](#) nella AWS AppSync Events Developer Guide.

In AWS AppSync, puoi annullare forzatamente l'iscrizione e chiudere (invalidare) una WebSocket connessione da un client connesso in base a una logica di filtro specifica. Ciò è utile in scenari relativi all'autorizzazione, ad esempio quando si rimuove un utente da un gruppo.

L'invalidazione dell'abbonamento si verifica in risposta a un payload definito in una mutazione. Ti consigliamo di considerare le mutazioni utilizzate per invalidare le connessioni di sottoscrizione come operazioni amministrative nell'API e di definire di conseguenza le autorizzazioni limitandone l'uso a un utente amministratore, un gruppo o un servizio di backend. Ad esempio, utilizzando direttive di autorizzazione dello schema come `@aws_auth(cognito_groups: ["Administrators"])` `@aws_iam`. Per ulteriori informazioni, vedere [Utilizzo di modalità di autorizzazione aggiuntive](#).

I filtri di invalidazione utilizzano la stessa sintassi e logica dei filtri di sottoscrizione [avanzati](#). Definite questi filtri utilizzando le seguenti utilità:

- `extensions.invalidateSubscriptions()`— Definito nel gestore di risposte del resolver GraphQL per una mutazione.
- `extensions.setSubscriptionInvalidationFilter()`— Definito nel gestore delle risposte del resolver GraphQL degli abbonamenti collegati alla mutazione.

[Per ulteriori informazioni sulle estensioni di filtro di invalidazione, consulta la panoramica dei resolver. JavaScript](#)

Utilizzo dell'invalidazione dell'abbonamento

Per vedere come funziona l'invalidazione dell'abbonamento AWS AppSync, usa il seguente schema GraphQL:

```
type User {
```

```

    userId: ID!
    groupId: ID!
  }

  type Group {
    groupId: ID!
    name: String!
    members: [ID!]!
  }

  type GroupMessage {
    userId: ID!
    groupId: ID!
    message: String!
  }

  type Mutation {
    createGroupMessage(userId: ID!, groupId : ID!, message: String!): GroupMessage
    removeUserFromGroup(userId: ID!, groupId : ID!) : User @aws_iam
  }

  type Subscription {
    onGroupMessageCreated(userId: ID!, groupId : ID!): GroupMessage
      @aws_subscribe(mutations: ["createGroupMessage"])
  }

  type Query {
    none: String
  }

```

Definisci un filtro di invalidazione nel codice del mutation resolver: `removeUserFromGroup`

```

import { extensions } from '@aws-appsync/utils';

export function request(ctx) {
  return { payload: null };
}

export function response(ctx) {
  const { userId, groupId } = ctx.args;
  extensions.invalidateSubscriptions({
    subscriptionField: 'onGroupMessageCreated',
    payload: { userId, groupId },

```

```
});
return { userId, groupId };
}
```

Quando viene richiamata la mutazione, i dati definiti nell'payload oggetto vengono utilizzati per annullare l'iscrizione definita in `subscriptionField`. Un filtro di invalidazione è inoltre definito nel modello di mappatura delle risposte dell'`onGroupMessageCreated` abbonamento.

Se il `extensions.invalidateSubscriptions()` payload contiene un ID che corrisponde a quello del IDs client sottoscritto come definito nel filtro, l'abbonamento corrispondente viene annullato. Inoltre, la connessione è chiusa. WebSocket Definisci il codice del resolver di sottoscrizione per l'`onGroupMessageCreated` abbonamento:

```
import { util, extensions } from '@aws-appsync/utils';

export function request(ctx) {
  // simplify return null for the payload
  return { payload: null };
}

export function response(ctx) {
  const filter = { groupId: { eq: ctx.args.groupId } };
  extensions.setSubscriptionFilter(util.transform.toSubscriptionFilter(filter));

  const invalidation = { groupId: { eq: ctx.args.groupId }, userId: { eq:
    ctx.args.userId } };
  extensions.setSubscriptionInvalidationFilter(util.transform.toSubscriptionFilter(invalidation));

  return null;
}
```

Tieni presente che il gestore della risposta alla sottoscrizione può avere sia filtri di sottoscrizione che filtri di invalidazione definiti contemporaneamente.

Ad esempio, supponiamo che il client A iscriva un nuovo utente con l'ID *user-1* al gruppo con l'ID *group-1* utilizzando la seguente richiesta di iscrizione:

```
onGroupMessageCreated(userId : "user-1", groupId : "group-1"){...}
```

AWS AppSync esegue il resolver di sottoscrizione, che genera filtri di sottoscrizione e invalidazione come definito nel precedente modello di mappatura delle risposte. `onGroupMessageCreated`

Per il client A, i filtri di sottoscrizione consentono l'invio dei dati solo *agroup-1*, mentre i filtri di invalidazione sono definiti per entrambi e. *user-1 group-1*

Supponiamo ora che il client B sottoscriva un utente con l'ID *user-2* a un gruppo con l'ID *group-2* utilizzando la seguente richiesta di iscrizione:

```
onGroupMessageCreated(userId : "user-2", groupId : "group-2") {...}
```

AWS AppSync esegue il resolver di sottoscrizione, che genera filtri di sottoscrizione e invalidazione.

Per il client B, i filtri di sottoscrizione consentono l'invio dei dati solo *agroup-2*, mentre i filtri di invalidazione sono definiti per entrambi e. *user-2 group-2*

Quindi, supponiamo che un nuovo messaggio di gruppo con l'ID *message-1* venga creato utilizzando una richiesta di mutazione come nell'esempio seguente:

```
createGroupMessage(id: "message-1", groupId :  
  "group-1", message: "test message") {...}
```

I client abbonati che corrispondono ai filtri definiti ricevono automaticamente il seguente payload di dati tramite: WebSockets

```
{  
  "data": {  
    "onGroupMessageCreated": {  
      "id": "message-1",  
      "groupId": "group-1",  
      "message": "test message",  
    }  
  }  
}
```

Il client A riceve il messaggio perché i criteri di filtraggio corrispondono al filtro di sottoscrizione definito. Tuttavia, il client B non riceve il messaggio, in quanto l'utente non ne fa parte *group-1*. Inoltre, la richiesta non corrisponde al filtro di sottoscrizione definito nel resolver di sottoscrizione.

Infine, supponiamo che *user-1* venga rimosso dall'*group-1* utilizzando la seguente richiesta di mutazione:

```
removeUserFromGroup(userId: "user-1", groupId : "group-1") {...}
```

La mutazione avvia un'invalidazione dell'abbonamento come definito nel codice del gestore della risposta del resolver. `extensions.invalidateSubscriptions()` AWS AppSync quindi annulla l'iscrizione al client A e chiude la sua connessione. WebSocket Il client B non è interessato, poiché il payload di invalidazione definito nella mutazione non corrisponde al suo utente o gruppo.

Quando AWS AppSync invalida una connessione, il client riceve un messaggio che conferma che l'iscrizione è stata annullata:

```
{
  "message": "Subscription complete."
}
```

Utilizzo di variabili di contesto nei filtri di invalidazione delle sottoscrizioni

Come per i filtri di sottoscrizione avanzati, puoi utilizzare la [contextvariable](#) nell'estensione del filtro di invalidazione dell'abbonamento per accedere a determinati dati.

Ad esempio, è possibile configurare un indirizzo e-mail come payload di invalidazione nella mutazione, quindi confrontarlo con l'attributo email o la dichiarazione di un utente abbonato autorizzato con i pool di utenti di Amazon Cognito o OpenID Connect. Il filtro di invalidazione definito nell'invalidatore dell'`extensions.setSubscriptionInvalidationFilter()` abbonamento verifica se l'indirizzo e-mail impostato dal `extensions.invalidateSubscriptions()` payload della mutazione corrisponde all'indirizzo e-mail recuperato dal token JWT dell'utente `context.identity.claims.email`, avviando l'invalidazione.

WebSocket Creazione di un client in tempo reale in AWS AppSync

Important

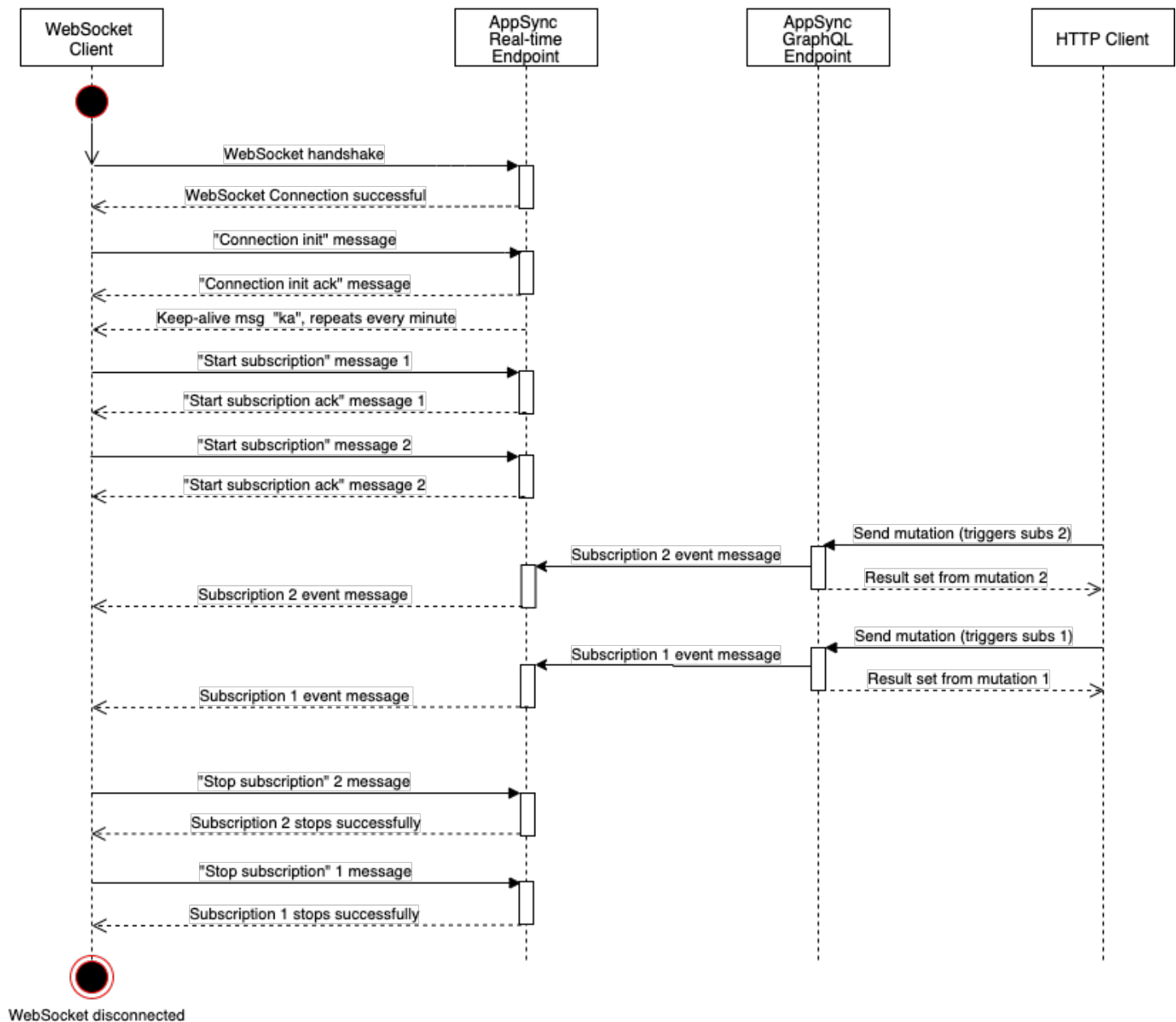
A partire dal 13 marzo 2025, puoi creare un' PubSub API in tempo reale basata su WebSockets AWS AppSync Events. Per ulteriori informazioni, consulta [Pubblicare eventi tramite WebSocket](#) nella AWS AppSync Events Developer Guide.

AWS AppSync il WebSocket client in tempo reale abilita gli abbonamenti GraphQL attraverso un processo in più fasi. Il client stabilisce innanzitutto una WebSocket connessione con l'endpoint AWS AppSync in tempo reale, invia un messaggio di inizializzazione della connessione e attende la conferma. Dopo una connessione riuscita, il client registra gli abbonamenti inviando messaggi

di avvio con query uniche e IDs GraphQL. AWS AppSync conferma gli abbonamenti eseguiti con successo con messaggi di conferma. Il client ascolta quindi gli eventi di sottoscrizione, che vengono attivati dalle mutazioni corrispondenti. Per mantenere la connessione, AWS AppSync invia messaggi keep-alive periodici. Al termine, il client annulla la registrazione degli abbonamenti inviando messaggi di arresto. Questo sistema supporta più abbonamenti su una singola WebSocket connessione e supporta varie modalità di autorizzazione, tra cui chiavi API, pool di utenti Amazon Cognito, IAM e Lambda.

Implementazione WebSocket client in tempo reale per gli abbonamenti GraphQL

Il diagramma di sequenza e i passaggi seguenti mostrano il flusso di lavoro degli abbonamenti in tempo reale tra il WebSocket client, il client HTTP e AWS AppSync



1. Il client stabilisce una WebSocket connessione con l'endpoint in tempo reale. AWS AppSync Se si verifica un errore di rete, il client dovrebbe eseguire un backoff esponenziale jittered. Per ulteriori informazioni, consulta [Exponential backoff and jitter](#) sul blog di architettura. AWS
2. (Facoltativo) Dopo aver stabilito correttamente la WebSocket connessione, il client invia un messaggio. `connection_init`
3. Se `connection_init` viene inviato, il client attende un `connection_ack` messaggio da AWS AppSync. Questo messaggio include un `connectionTimeoutMs` parametro, che è il tempo di attesa massimo in millisecondi per un messaggio "ka" (keep-alive).

4. AWS AppSync "ka" invia messaggi periodicamente. Il client tiene traccia dell'ora in cui ha ricevuto ogni "ka" messaggio. Se il client non riceve un "ka" messaggio entro `connectionTimeoutMs` millisecondi, deve chiudere la connessione.
5. Il client registra la sottoscrizione inviando un messaggio di sottoscrizione `start`. Una singola WebSocket connessione supporta più abbonamenti, anche se sono in modalità di autorizzazione diverse.
6. Il client attende l'invio di `start_ack` messaggi AWS AppSync per confermare l'avvenuta sottoscrizione. Se c'è un errore, AWS AppSync restituisce un `"type": "error"` messaggio.
7. Il client ascolta gli eventi di sottoscrizione, che vengono inviati dopo la chiamata di una mutazione corrispondente. Le interrogazioni e le mutazioni vengono generalmente inviate `https://` all'endpoint GraphQL AWS AppSync. Le sottoscrizioni fluiscono attraverso l'endpoint AWS AppSync in tempo reale utilizzando `secure()`. WebSocket `wss://`
8. Il client annulla la registrazione della sottoscrizione inviando un messaggio di sottoscrizione `stop`.
9. Dopo aver annullato la registrazione di tutti gli abbonamenti e aver verificato che non vi siano messaggi trasferiti tramite il WebSocket, il client può disconnettersi dalla connessione. WebSocket

Stabilisci i dettagli della stretta di mano per stabilire la connessione WebSocket

Per connettersi e avviare una stretta di mano di successo con AWS AppSync, un WebSocket client deve disporre di quanto segue:

- L' AWS AppSync endpoint in tempo reale
- Intestazioni: contengono informazioni relative all' AWS AppSync endpoint e all'autorizzazione. AWS AppSync supporta i seguenti tre metodi per fornire le intestazioni:
 - Intestazioni tramite stringa di query
 - Le informazioni sull'intestazione sono codificate come una stringa base64, derivata da un oggetto JSON con stringhe. Questo oggetto JSON contiene dettagli relativi all'endpoint e all'autorizzazione. AWS AppSync Il contenuto dell'oggetto JSON varia a seconda della modalità di autorizzazione.
 - Intestazioni tramite `Sec-WebSocket-Protocol`
 - Una stringa con codifica Base64URL proveniente dall'oggetto JSON con stringhe che contiene informazioni relative all' AWS AppSync endpoint e all'autorizzazione viene passata come protocollo nell'intestazione. `Sec-WebSocket-Protocol` Il contenuto dell'oggetto JSON varia a seconda della modalità di autorizzazione.

- Intestazioni tramite intestazioni HTTP standard:
 - Le intestazioni possono essere passate come intestazioni HTTP standard nella richiesta di connessione, in modo simile a come vengono passate le intestazioni per le query e le mutazioni GraphQL. AWS AppSync Tuttavia, il passaggio di intestazioni tramite intestazioni HTTP standard non è supportato per le richieste di connessione API private.
- payload— Stringa codificata in Base64 di. payload Il payload è necessario solo se le intestazioni vengono fornite utilizzando la stringa di query

Con questi requisiti, un WebSocket client può connettersi all'URL, che contiene l'endpoint in tempo reale con la stringa di query, utilizzato graphql-ws come protocollo. WebSocket

Rilevamento dell'endpoint in tempo reale dall'endpoint GraphQL

L'endpoint AWS AppSync GraphQL e l'endpoint AWS AppSync in tempo reale sono leggermente diversi nel protocollo e nel dominio. È possibile recuperare l'endpoint GraphQL utilizzando AWS Command Line Interface il AWS CLI comando (). `aws appsync get-graphql-api`

AWS AppSync Endpoint GraphQL:

```
https://example1234567890000.apps-sync-api.us-east-1.amazonaws.com/graphql
```

AWS AppSync endpoint in tempo reale:

```
wss://example1234567890000.apps-sync-realtime-api.us-east-1.amazonaws.com/graphql
```

Le applicazioni possono connettersi all'endpoint AWS AppSync GraphQL (`https://`) utilizzando qualsiasi client HTTP per query e mutazioni. Le applicazioni possono connettersi all'endpoint AWS AppSync in tempo reale (`wss://`) utilizzando qualsiasi client per le sottoscrizioni. WebSocket

Con i nomi di dominio personalizzati, puoi interagire con entrambi gli endpoint utilizzando un unico dominio. Ad esempio, se configuri `api.example.com` come dominio personalizzato, puoi interagire con GraphQL e gli endpoint in tempo reale utilizzando questi: URLs

AWS AppSync endpoint GraphQL di dominio personalizzato:

```
https://api.example.com/graphql
```

AWS AppSync endpoint in tempo reale con dominio personalizzato:

```
wss://api.example.com/graphql/realtime
```

Formato dei parametri di intestazione basato sulla modalità di autorizzazione AWS AppSync API

Il formato dell'header oggetto utilizzato nella stringa di query di connessione varia a seconda della modalità di autorizzazione dell' AWS AppSync API. Il `host` campo nell'oggetto si riferisce all'endpoint AWS AppSync GraphQL, che viene utilizzato per convalidare la connessione anche se la `wss://` chiamata viene effettuata sull'endpoint in tempo reale. Per avviare l'handshake e stabilire la connessione autorizzata, `payload` deve essere un oggetto JSON vuoto. Il payload è necessario solo se le intestazioni vengono passate tramite una stringa di query.

Le sezioni seguenti mostrano i formati di intestazione per ciascuna modalità di autorizzazione.

Chiave API

Intestazione della chiave API

Contenuto dell'intestazione

- `"host": <string>`: l'host per l'endpoint AWS AppSync GraphQL o il nome di dominio personalizzato.
- `"x-api-key": <string>`: la chiave API configurata per l' AWS AppSync API.

Esempio

```
{
  "host": "example1234567890000.appsync-api.us-east-1.amazonaws.com",
  "x-api-key": "da2-12345678901234567890123456"
}
```

Intestazioni tramite stringa di query

Innanzitutto, un oggetto JSON contenente `host` and the `x-api-key` viene convertito in una stringa. Successivamente, questa stringa viene codificata utilizzando la codifica base64. La stringa risultante con codifica base64 viene aggiunta come parametro di query denominato all' `WebSocketURL` per stabilire la connessione con l'endpoint `header` in tempo reale. AWS AppSync L'URL della richiesta risultante assume il formato seguente:

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql?
header=eyJ0b3N0IjoiaXh0bXBsZTEyMzQ1Njc4OTAwMDAuYXBwc3luYy1hcGkudXMtZWZzdC0xLmFtYXpvbmF3cy5jb20i
```

È importante notare che oltre all'oggetto header con codifica base64, anche un oggetto JSON vuoto {} è codificato in base64 e incluso come parametro di query separato denominato nell'URL. payload WebSocket

Intestazioni tramite **Sec-WebSocket-Protocol**

Un oggetto JSON contenente host and the x-api-key viene convertito in una stringa e quindi codificato utilizzando la codifica Base64url. La stringa con codifica Base64URL risultante ha il prefisso. header- Questa stringa con prefisso viene quindi utilizzata come nuovo sottoprotocollo oltre che nell'Sec-WebSocket-Protocol intestazione quando si stabilisce la connessione con l'graphql-ws endpoint in tempo reale. WebSocket AWS AppSync

L'URL della richiesta risultante assume il seguente formato:

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql
```

L'Sec-WebSocket-Protocol intestazione contiene il seguente valore:

```
"sec-websocket-protocol" : ["graphql-ws", "header-  
ewogICAgImhvc3QiOiJleGFtcGx1MTIzNDU2Nzg5MDAwMC5hcHBzeW5jLWFWaS51cy1lYXN0LTEuYW1hem9uYXdzLmNvbSI
```

Intestazioni tramite intestazioni HTTP standard

In questo metodo, le informazioni sull'host e sulla chiave API vengono trasmesse utilizzando intestazioni HTTP standard quando si stabilisce la WebSocket connessione con l' AWS AppSync endpoint in tempo reale. L'URL della richiesta risultante assume il seguente formato:

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql
```

Le intestazioni della richiesta includerebbero quanto segue:

```
"sec-websocket-protocol" : ["graphql-ws"]  
"host": "example1234567890000.appsync-api.us-east-1.amazonaws.com",  
"x-api-key": "da2-12345678901234567890123456"
```

Pool di utenti Amazon Cognito e OpenID Connect (OIDC)

Amazon Cognito e header OIDC

Contenuti dell'intestazione:

- "Authorization": <string>: Un token ID JWT. L'intestazione può utilizzare uno schema [Bearer](#).
- "host": <string>: l'host per l'endpoint AWS AppSync GraphQL o il nome di dominio personalizzato.

Esempio:

```
{
  "Authorization": "eyJEXAMPLEiJjbG5xb3A5eW5MK09QYXIrMTJHWEFLSXBieU5WNHhsQjEXAMPLEnM2WldvPSIsImFsZS8zZW5kIjE0L2Z3Yi7f-SmEGoh2AD8emxQRYajByz-rE4Jh0Q0ymN2Ys-ZIKMpVBTPgu-TMWdy0HhDUMUj20P82yeZ3w1Zatr_gM4LzjXUXmI_K2yGjuXfXTaa1mvQEBG0mQfVd7SfwXB-jcv4RYVi6j25qgow9Ew52ufurPqaK-3WAKG32KpV8J4-Wejq8t0c-yA7sb8EnB551b7TU93uKRiVVK3E55Nk5ADPoam_WYE45i3s5qVAP_-InW75NUo0CGTsS8YWMfb6ecHYJ-1j-bzA27zaT9VjctXn9byNFZmEXAMPLExw",
  "host": "example1234567890000.appsync-api.us-east-1.amazonaws.com"
}
```

Intestazioni tramite stringa di query

Innanzitutto, un oggetto JSON contenente host and the Authorization viene convertito in una stringa. Successivamente, questa stringa viene codificata utilizzando la codifica base64. La stringa risultante con codifica base64 viene aggiunta come parametro di query denominato all' WebSocket URL per stabilire la connessione con l'endpoint header in tempo reale. AWS AppSync L'URL della richiesta risultante assume il formato seguente:

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql?
header=eyJBdXRob3JpemF0aW9uIjoiZXlKcmFXUWlPaUpqYkc1eGIzQTVlVzVNSzA5UVlYSXJNVEpIV0VGTFNYQm1lVTVX
```

È importante notare che oltre all'oggetto header con codifica base64, anche un oggetto JSON vuoto {} è codificato in base64 e incluso come parametro di query separato denominato nell'URL. payload WebSocket

Intestazioni tramite **Sec-WebSocket-Protocol**

Un oggetto JSON contenente host and the Authorization viene convertito in una stringa e quindi codificato utilizzando la codifica Base64url. La stringa con codifica Base64URL risultante ha il prefisso. header - Questa stringa con prefisso viene quindi utilizzata come nuovo sottoprotocollo

oltre che nell'`Sec-WebSocket-Protocol` intestazione quando si stabilisce la connessione con l'`graphql-ws` endpoint in tempo reale. WebSocket AWS AppSync

L'URL della richiesta risultante assume il seguente formato:

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql
```

L'`Sec-WebSocket-Protocol` intestazione contiene il seguente valore:

```
"sec-websocket-protocol" : ["graphql-ws", "header-  
ewogICAgImhvc3QiOiJleGFtcGx1MTIzNDU2Nzg5MDAwMCM5hChBzeW5jLWFwaS51cy11YXN0LTEuYW1hem9uYXdzLmNvbSI
```

Intestazioni tramite intestazioni HTTP standard

In questo metodo, le informazioni sull'host e sull'autorizzazione vengono trasmesse utilizzando intestazioni HTTP standard quando si stabilisce la WebSocket connessione con l' AWS AppSync endpoint in tempo reale. L'URL della richiesta risultante assume il seguente formato:

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql
```

Le intestazioni della richiesta includerebbero quanto segue:

```
"sec-websocket-protocol" : ["graphql-ws"]  
"Authorization": "eyJEXAMPLEiJjbG5xb3A5eW5MK09QYXIrMTJHWEFLSXBieU5WNHhsQjEXAMPLEnM2WldvPSIsImFsZy  
zEE2DJH7sH012zxYi7f-SmEGoh2AD8emxQRYajByz-rE4Jh0Q0ymN2Ys-ZIkMpVBTPgu-  
TMWdy0HhDumUj20P82yeZ3w1Zatr_gM4LzjXUXmI_K2yGjuXfXTaa1mvQEBG0mQfVd7SfwXB-  
jcv4RYVi6j25qgow9Ew52ufurPqaK-3WAKG32KpV8J4-Wejq8t0c-  
yA7sb8EnB551b7TU93uKRiVVK3E55Nk5ADPoam_WYE45i3s5qVAP_-InW75NUo0CGTsS8YWMfb6ecHYJ-1j-  
bzA27zaT9VjctXn9byNFZmEXAMPLExw",  
"host": "example1234567890000.appsync-api.us-east-1.amazonaws.com"
```

IAM

Intestazione IAM

Contenuto dell'intestazione

- `"accept": "application/json, text/javascript"`: un parametro `<string>` costante.
- `"content-encoding": "amz-1.0"`: un parametro `<string>` costante.
- `"content-type": "application/json; charset=UTF-8"`: un parametro `<string>` costante.

- "host": <string>: Questo è l'host per l'endpoint AWS AppSync GraphQL.
- "x-amz-date": <string>: Il timestamp deve essere in UTC e nel seguente formato ISO 8601: YYYYMMDD'T'HHMMSS'Z'. Ad esempio, 20150830T123600Z è un timestamp valido. Non includere i millisecondi nel time stamp. Per ulteriori informazioni, [vedere Gestione delle date Riferimenti generali di AWS](#) nella versione 4 di Signature in.
- "X-Amz-Security-Token": <string>: Il token di AWS sessione, necessario quando si utilizzano credenziali di sicurezza temporanee. Per ulteriori informazioni, consulta [Utilizzo di credenziali temporanee con le risorse AWS](#) nella Guida per l'utente IAM.
- "Authorization": <string>: Signature Version 4 (SigV4): informazioni di firma per l'endpoint. AWS AppSync Per ulteriori informazioni sul processo di firma, vedere [Attività 4: aggiungere la firma alla richiesta HTTP](#) in. Riferimenti generali di AWS

La richiesta HTTP di firma Sigv4 include un URL canonico, che è l'endpoint AWS AppSync GraphQL con /connect aggiunto. La AWS regione dell'endpoint del servizio è la stessa regione in cui stai utilizzando l' AWS AppSync API e il nome del servizio è «appsync». La richiesta HTTP da firmare è la seguente:

```
{
  url: "https://example1234567890000.apps-sync-api.us-east-1.amazonaws.com/graphql/
connect",
  data: "{}",
  method: "POST",
  headers: {
    "accept": "application/json, text/javascript",
    "content-encoding": "amz-1.0",
    "content-type": "application/json; charset=UTF-8",
  }
}
```

Esempio

```
{
  "accept": "application/json, text/javascript",
  "content-encoding": "amz-1.0",
  "content-type": "application/json; charset=UTF-8",
  "host": "example1234567890000.apps-sync-api.us-east-1.amazonaws.com",
  "x-amz-date": "20200401T001010Z",
  "X-Amz-Security-Token":
  "AgEXAMPLEZ2luX2VjEAoaDmFwLXNvdXR0ZWFEXAMPLECwRQIgAh97C1jq7w0PL8KsxP3YtDuyC/9hAj8PhJ7Fvf38SgoC"
```

```
+
+pEagWCveZUjKE0zyUhBEXAMPLEjj//////////8BEXAMPLEx0Dk2NDgyNzg1NSIMo1mWnpESWUoYw4BkKqEFSrm3DXuL8
+ZbVc4JKjDP4vUCKNR6Le9C9pZp9Psw0NoFy3vLBudAXEXAMPLE0VG8feXfiEEA+1khgFK/
wEtWR+9zF7NaMMmSe07wN2gG2tH0eKMEXAMPLEQX+sMbytQo8iepP9PZ0z1ZsSFb/
dP5Q8hk6YEXAMPLEYcKZsTkDAq2uKFQ8mYUVA9EtQnNRiFLEY83aKvG/tqLWNnG1SNVx7SMcfovkFDqQamm
+88y10wwAEYK7qcoceX6Z7GGcaYuIfGpaX2MCCELeQvZ+8WxEg0nIfz7GYvsYNjLZSaRnV4G
+ILY1F0QNW64S9Nvj
+BwDg3ht2CrNvpwjVY1j9U3nmxE0UG5ne83LL5hhqMpm25kmL7enVgw2kQzmU2id4IKu0C/
WaoDRu02F5zE63vJbxN8AYs7338+4B4HbB6BZ60Ugg96Q15RA41/
gIqxaVPxyTpDfTU5GfSLxocdYeniqqpFMtZG2n9d0u7GsQNcFkNcG3qDZm4tDo8tZbuym0a2VcF2E5hFEgXBa
+XLJCfXi/770qAEjP0x7Qdk3B43p8KG/BaioP5RsV8zBGvH1zAgyPha2rN70/
tT13yrmPd5QYEFwzexjKrV4mWIuRg8NTHYSZJUaeyCwTom80VFUJXG
+GYTUyv5W22aBcnoRGiCiKEYTL0kgXecdKFTHmcIAejQ9We1r0a196Kq87w5KNMCKcCGFnwBNFLmfmbpNqT6rUBxxs3X5nt
aox0FtHX21eF6qIGT8j1z+l2opU+ggwUgkhUUgCH2TfqBj+MLMVvpgqJsPKt582caFKArIFIv0
+9QupxLnEH2hz04TMTfnU6bQC6z1buVe7h
+t0Lnh1YPFsLQ88anib/7TTC8k9DsBTq0ASE8R2GbSEsm09qbbMwgEaYUh0KtGeyQsSJdhSk6XxxThrWL9EnwBCXDkICMqd
+WgtPtK00weDlCaRs3R2qXcbNgVhleMk4IwnF8D1695AenU1LwHj0JLkCjxgNFfiWAFEPH9aEXAMPLExA="",
  "Authorization": "AWS4-HMAC-SHA256 Credential=XXXXXXXXXXXXXXXXXX/20200401/
us-east-1/appsync/aws4_request, SignedHeaders=accept;content-
encoding;content-type;host;x-amz-date;x-amz-security-token,
  Signature=83EXAMPLEbcc1fe3ee69f75cd5ebbf4cb4f150e4f99cec869f149c5EXAMPLEdc"
}
```

Intestazioni tramite stringa di query

Innanzitutto, un oggetto JSON contenente host (endpoint AWS AppSync GraphQL) e le altre intestazioni di autorizzazione viene convertito in una stringa. Successivamente, questa stringa viene codificata utilizzando la codifica base64. La stringa con codifica base64 risultante viene aggiunta all'URL come parametro di query denominato `header`. L'URL della richiesta risultante assume il formato seguente:

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql?
header=eyJBdXRob3JpemF0aW9uIjoizXlKcmFXUWlPaUpqYkc1eGIzQTVlVzVNSzA5UVlYSXJNVEpIV0VGTFNYQm11VTVX
```

È importante notare che oltre all'oggetto header con codifica base64, anche un oggetto JSON vuoto `{}` è codificato in base64 e incluso come parametro di query separato denominato nell'URL. `payload` WebSocket

Intestazioni tramite **Sec-WebSocket-Protocol**

Un oggetto JSON contenente le host e le altre intestazioni di autorizzazione viene convertito in una stringa e quindi codificato utilizzando la codifica Base64URL. La stringa con codifica Base64URL

risultante ha il prefisso. `header-` Questa stringa con prefisso viene quindi utilizzata come nuovo sottoprotocollo oltre che nell'`Sec-WebSocket-Protocol` intestazione quando si stabilisce la connessione con l'`graphql-ws` endpoint in tempo reale. `WebSocket AWS AppSync`

L'URL della richiesta risultante assume il seguente formato:

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql
```

L'`Sec-WebSocket-Protocol` intestazione contiene il seguente valore:

```
"sec-websocket-protocol" : ["graphql-ws", "header-  
ew0KICAiYWNjZXB0IjogImFwcGxpY2F0aW9uL2pzb24sIHRleHQvamF2YXNjcm1wdCIuDQogICJjb250ZW50LWVuY29kaW50"]
```

Intestazioni tramite intestazioni HTTP standard

In questo metodo, l'`host` e le altre informazioni di autorizzazione vengono trasmesse utilizzando intestazioni HTTP standard quando si stabilisce la `WebSocket` connessione con l' `AWS AppSync` endpoint in tempo reale. L'URL della richiesta risultante assume il seguente formato:

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql
```

Le intestazioni della richiesta includerebbero quanto segue:

```
"sec-websocket-protocol" : ["graphql-ws"]  
"accept": "application/json, text/javascript",  
"content-encoding": "amz-1.0",  
"content-type": "application/json; charset=UTF-8",  
"host": "example1234567890000.appsync-api.us-east-1.amazonaws.com",  
"x-amz-date": "20200401T001010Z",  
"X-Amz-Security-Token":  
  "AgEXAMPLEZ2luX2VjEAoaDmFwLXNvdXR0ZWFEEXAMPLEEcwRQIgAh97Cljq7w0PL8KsxP3YtDuyC/9hAj8PhJ7Fvf38SgoC  
  +  
  +pEagWCveZUjKE0zyUhBEXAMPLEjj/////////8BEXAMPLEx0Dk2NDgyNzg1NSIMo1mWnpESWUoYw4BkKqEFS1m3DXuL8  
  +ZbVc4JKjDP4vUCKNR6Le9C9pZp9PsW0NoFy3vLBUDAXEXAMPLE0VG8feXfiEEA+1khgFK/  
  wEtwR+9zF7NaMMmSe07wN2gG2tH0eKMEXAMPLEQX+sMbytQo8iepP9PZ0z1ZsSFb/  
  dP5Q8hk6YEXAMPLEYcKZsTkDAq2uKFQ8mYUVA9EtQnNRiFLEY83aKvG/tqLWNnG1SNVx7SMcfovkFDqQamm  
  +88y10wwAEYK7qcocex6Z7GGcaYuIfGpaX2MCCELeQvZ+8WxEg0nIfz7GYvsYNjLZSaRnV4G  
  +ILY1F0QNW64S9Nvj  
  +BwDg3ht2CrNvpwjVY1j9U3nmxE0UG5ne83LL5hhqMpm25kmL7enVgw2kQzmU2id4IKu0C/  
  WaoDRu02F5zE63vJbxN8AYs7338+4B4Hb6BZ60Ugg96Q15RA41/  
  gIqxaVPxyTpDfTU5GfSLxocdYeniqqpFMtZG2n9d0u7GsQNCfKNCg3qDZm4tDo8tZbuym0a2VcF2E5hFEgXBa  
  +XLJCfXi/770qAEjP0x7Qdk3B43p8KG/BaioP5RsV8zBGvH1zAgyPha2rN70/
```

```
tT13yrmPd5QYefwzexjKrV4mWIuRg8NTHYSZJUaeyCwTom80VFUJXG
+GYUyv5W22aBcnoRGiCiKEYTL0kgXecdKFTHmcIAejQ9We1r0a196Kq87w5KNMckcCGFnwBNFLmfmbpNqT6rUBxss3X5nt
aox0FtHX21eF6qIGT8j1z+l2opU+ggwUgkhUUgCH2TfqBj+MLMVVvpqgJsPKt582caFKArIFiv0
+9QupxLnEH2hz04TMTfnU6bQC6z1buVe7h
+t0Lnh1YPFsLQ88anib/7TTC8k9DsBTq0ASe8R2GbSEsm09qbbMwgEaYUh0KtGeyQsSJdhSk6XxXThrWL9EnwBCXDkICMqd
+WgtPtK00weDlCaRs3R2qXcbNgVhleMk4IWnF8D1695AenU1LwHj0JLkCjxgNFfiWAFEPH9aEXAMPLExA==" ,
"Authorization": "AWS4-HMAC-SHA256 Credential=XXXXXXXXXXXXXXXXXXXX/20200401/
us-east-1/appsycn/aws4_request, SignedHeaders=accept;content-
encoding;content-type;host;x-amz-date;x-amz-security-token,
Signature=83EXAMPLEebcc1fe3ee69f75cd5ebbf4cb4f150e4f99cec869f149c5EXAMPLEdc"
```

Per firmare la richiesta utilizzando un dominio personalizzato:

```
{
  url: "https://api.example.com/graphql/connect",
  data: "{}",
  method: "POST",
  headers: {
    "accept": "application/json, text/javascript",
    "content-encoding": "amz-1.0",
    "content-type": "application/json; charset=UTF-8",
  }
}
```

Esempio

```
{
  "accept": "application/json, text/javascript",
  "content-encoding": "amz-1.0",
  "content-type": "application/json; charset=UTF-8",
  "host": "api.example.com",
  "x-amz-date": "20200401T001010Z",
  "X-Amz-Security-Token":
    "AgEXAMPLEZ2luX2VjEAoaDmFwLXNvdXR0ZWFEXAMPLECwRQIgAh97Cljq7w0PL8Ksxp3YtDuyC/9hAj8PhJ7Fvf38SgoC
    +
    +pEagWCveZUjKE0zyUhBEXAMPLEjj/////////8BEXAMPLEx0Dk2NDgyNzg1NSIMo1mWnpESWUoYw4BkKqEFS1m3DXuL8
    +ZbVc4JKjDP4vUCKNR6Le9C9pZp9PsW0NoFy3vLBUDAXEXAMPLE0VG8feXfiEEA+1khgFK/
    wEtW+9zF7NaMMmSe07wN2gG2tH0eKMEXAMPLEEQX+sMbytQo8iepP9PZ0z1ZsSFb/
    dP5Q8hk6YEXAMPLEYcKZsTkDAq2uKFQ8mYUVA9EtQnNRiFLEY83aKvG/tqLWNnG1SNVx7SMcfovkFDqQamm
    +88y10wwAEYK7qcocex6Z7GGcaYuIfGpaX2MCCELeQvZ+8WxEg0nIfz7GYvsYNjLZSaRnV4G
    +ILY1F0QNW64S9Nvj
    +BwDg3ht2CrNvpwjVY1j9U3nmxE0UG5ne83LL5hhqMpm25kmL7enVgw2kQzmU2id4IKu0C/
    WaoDRu02F5zE63vJbxN8AYs7338+4B4HBb6BZ60Ugg96Q15RA41/
```

```
gIqxaVPxyTpDfTU5GfSLxocdYeniqqpFMtZG2n9d0u7GsQNcFkNcG3qDZm4tDo8tZbuym0a2VcF2E5hFEgXBa
+XLJCfXi/770qAEjP0x7Qdk3B43p8KG/BaioP5RsV8zBGvH1zAgyPha2rN70/
tT13yrmPd5QYefwzexjKrV4mWIuRg8NTHYSZJUaeyCwTom80VFUJXG
+GYTUyv5W22aBcnoRGiCiKEYTL0kgXecdKFTHmcIAejQ9We1r0a196Kq87w5KNMckcCGFnwBNFLmfmbpNqT6rUBxxs3X5nt
aox0FtHX21eF6qIGT8j1z+12opU+ggwUgkhUUgCH2TfqBj+MLMVVvpqgJsPKt582caFKArIFiv0
+9QupxLnEH2hz04TMTfnU6bQC6z1buVe7h
+t0Lnh1YPFsLQ88anib/7TTC8k9DsBTq0ASe8R2GbSEsm09qbbMwgEaYUh0KtGeyQsSJdhSk6XxXThrWL9EnwBCXDkICMqd
+WgtPtK00weDlCaRs3R2qXcbNgVhleMk4IwnF8D1695AenU1LwHj0JLkCjxgNFfiWAFEPH9aEXAMPLExA="",
  "Authorization": "AWS4-HMAC-SHA256 Credential=XXXXXXXXXXXXXXXXXXXX/20200401/
us-east-1/appsync/aws4_request, SignedHeaders=accept;content-
encoding;content-type;host;x-amz-date;x-amz-security-token,
Signature=83EXAMPLEebcc1fe3ee69f75cd5ebbf4cb4f150e4f99cec869f149c5EXAMPLEdc"
}
```

URL della richiesta con stringa di query

```
wss://api.example.com/graphql?
header=eyJEXAMPLEHQiOiJhcHBsaWNhdGlvbi9qc29uLCB0ZXh0L2phdmFEXAMPLEQiLCJjb250ZW50LWVuY29kaW5nIjoE
```

Note

Una WebSocket connessione può avere più abbonamenti (anche con diverse modalità di autenticazione). Un modo per implementarlo consiste nel creare una WebSocket connessione per il primo abbonamento e poi chiuderlo quando l'ultimo abbonamento non è registrato. Puoi ottimizzarlo attendendo qualche secondo prima di chiudere la WebSocket connessione, nel caso in cui l'app venga sottoscritta immediatamente dopo la cancellazione dell'ultimo abbonamento. Ad esempio, quando si passa da una schermata all'altra, quando si smonta un abbonamento si interrompe e, in caso di montaggio, si avvia un abbonamento diverso.

Autorizzazione Lambda

Intestazione di autorizzazione Lambda

Contenuto dell'intestazione

- "Authorization": <string>: Il valore che viene passato come `authorizationToken`.
- "host": <string>: l'host per l'endpoint AWS AppSync GraphQL o il nome di dominio personalizzato.

Esempio

```
{
  "Authorization": "M0UzQzM1MkQtMkI0Ni00OTZCLUi1NkQtMUM0MTQ0QjVBRTczCkI1REEzRTIxLTk5NzItNDJENiBQ",
  "host": "example1234567890000.appsync-api.us-east-1.amazonaws.com"
}
```

Intestazioni tramite stringa di query

Innanzitutto, un oggetto JSON contenente `host` and the `Authorization` viene convertito in una stringa. Successivamente, questa stringa viene codificata utilizzando la codifica `base64`. La stringa risultante con codifica `base64` viene aggiunta come parametro di query denominato all' `WebSocket` URL per stabilire la connessione con l'endpoint `header` in tempo reale. AWS AppSync L'URL della richiesta risultante assume il formato seguente:

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql?
header=eyJBdXRob3JpemF0aw9uIjoizXlKcmFXUWlPaUpqYkc1eGIzQTVlVzVNSzA5UVlYSXJNVEpIV0VGTfNYQmllVTXV
```

È importante notare che oltre all'oggetto header con codifica base64, anche un oggetto JSON vuoto {} è codificato in base64 e incluso come parametro di query separato denominato nell'URL. payload WebSocket

Intestazioni tramite **Sec-WebSocket-Protocol**

Un oggetto JSON contenente `host` and the `Authorization` viene convertito in una stringa e quindi codificato utilizzando la codifica `Base64url`. La stringa con codifica `Base64URL` risultante ha il prefisso `header-`. Questa stringa con prefisso viene quindi utilizzata come nuovo sottoprotocollo oltre che nell'`Sec-WebSocket-Protocol` intestazione quando si stabilisce la connessione con `graphql-ws` endpoint in tempo reale. `WebSocket` `AWS AppSync`

L'URL della richiesta risultante assume il seguente formato:

wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql

L'Sec-WebSocket-Protocolintestazione contiene il seguente valore:

```
"sec-websocket-protocol" : ["graphql-ws", "header-ewogICAgImhvc3QiOiJleGFtcGx1MTIzNDU2Nzg5MDAwMC5hcHBzeW5jLWFWaS51cy1lYXN0LTEuYW1hem9uYXdzLmNvbSI
```

Intestazioni tramite intestazioni HTTP standard

In questo metodo, le informazioni sull'host e sull'autorizzazione vengono trasmesse utilizzando intestazioni HTTP standard quando si stabilisce la WebSocket connessione con l' AWS AppSync endpoint in tempo reale. L'URL della richiesta risultante assume il seguente formato:

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql
```

Le intestazioni della richiesta includerebbero quanto segue:

```
"sec-websocket-protocol" : ["graphql-ws"]
"Authorization": "eyJEXAMPLEiJjbG5xb3A5eW5MK09QYXIrMTJHWEFLSXBieU5WNHhsQjEXAMPLEnM2WldvPSIsImFsZy9zEE2DJH7sH012zxYi7f-SmEGoh2AD8emxQRYajByz-rE4Jh0Q0ymN2Ys-ZIkMpVBTPgu-TMWdy0HhDumUj20P82yeZ3wLZAtr_gM4LzjXUXmI_K2yGjuXfXTaa1mvQEBG0mQfVd7SfwXB-jcv4RYVi6j25qgow9Ew52ufurPqaK-3WAKG32KpV8J4-Wejq8t0c-yA7sb8EnB551b7TU93uKRiVVK3E55Nk5ADPoam_WYE45i3s5qVAP_-InW75NUo0CGTsS8YWMfb6ecHYJ-1j-bzA27zaT9VjctXn9byNFZmEXAMPLExw",
"host": "example1234567890000.appsync-api.us-east-1.amazonaws.com"
```

Funzionamento in tempo reale WebSocket

Dopo aver avviato con successo una WebSocket stretta di mano con AWS AppSync, il client deve inviare un messaggio successivo a cui connettersi AWS AppSync per diverse operazioni. Questi messaggi richiedono i seguenti dati:

- **type**: il tipo di operazione.
- **id**: un identificatore univoco per l'abbonamento. Si consiglia di utilizzare un UUID per questo scopo.
- **payload**: Il payload associato, a seconda del tipo di operazione.

Il **type** campo è l'unico campo obbligatorio; i **payload** campi **id** e sono facoltativi.

Sequenza di eventi

Per avviare, stabilire, registrare ed elaborare correttamente la richiesta di iscrizione, il cliente deve seguire la seguente sequenza:

1. Inizializza connessione (`connection_init`)
2. Riconoscimento connessione (`connection_ack`)

3. Registrazione sottoscrizione (`start`)
4. Riconoscimento sottoscrizione (`start_ack`)
5. Elaborazione della sottoscrizione (`data`)
6. Annullamento sottoscrizione (`stop`)

Messaggio init di connessione

(Facoltativo) Dopo una stretta di mano riuscita, il client può inviare il `connection_init` messaggio per iniziare a comunicare con l' AWS AppSync endpoint in tempo reale. Il messaggio è una stringa ottenuta stringendo l'oggetto JSON come segue:

```
{ "type": "connection_init" }
```

Messaggio di conferma connessione

Dopo aver inviato il messaggio `connection_init`, il client deve attendere il messaggio `connection_ack`. Tutti i messaggi inviati prima della ricezione vengono ignorati. `connection_ack` Il messaggio dovrebbe essere letto come segue:

```
{
  "type": "connection_ack",
  "payload": {
    // Time in milliseconds waiting for ka message before the client should terminate
    the WebSocket connection
    "connectionTimeoutMs": 300000
  }
}
```

Messaggio keep-alive

Oltre al messaggio di conferma della connessione, il client riceve periodicamente messaggi keep-alive. Se il client non riceve un messaggio keep-alive entro il periodo di timeout della connessione, deve chiudere la connessione. AWS AppSync continua a inviare questi messaggi e a gestire gli abbonamenti registrati fino a quando non interrompe automaticamente la connessione (dopo 24 ore). I messaggi Keep-alive sono battiti cardiaci e non è necessario che il client li riconosca.

```
{ "type": "ka" }
```


Messaggio di registrazione dell'abbonamento

Dopo aver ricevuto un `connection_ack` messaggio, il client può inviare messaggi di registrazione dell'abbonamento a. AWS AppSync Questo tipo di messaggio è un oggetto JSON con stringhe che contiene i seguenti campi:

- `"id"`: `<string>`: L'ID dell'abbonamento. Questo ID deve essere univoco per ogni abbonamento, altrimenti il server restituisce un errore che indica che l'ID dell'abbonamento è duplicato.
- `"type"`: `"start"`: un parametro `<string>` costante.
- `"payload"`: `<Object>`: Un oggetto che contiene le informazioni relative all'abbonamento.
 - `"data"`: `<string>`: un oggetto JSON con stringhe che contiene una query GraphQL e variabili.
 - `"query"`: `<string>`: Un'operazione GraphQL.
 - `"variables"`: `<Object>`: Un oggetto che contiene le variabili per la query.
 - `"extensions"`: `<Object>`: Un oggetto che contiene un oggetto di autorizzazione.
- `"authorization"`: `<Object>`: Un oggetto che contiene i campi richiesti per l'autorizzazione.

Oggetto autorizzazione per la registrazione della sottoscrizione

Le stesse regole della [Formato dei parametri di intestazione basato sulla modalità di autorizzazione AWS AppSync API](#) sezione si applicano all'oggetto di autorizzazione. L'unica eccezione è per IAM, in cui le informazioni sulla firma SigV4 sono leggermente diverse. Per ulteriori dettagli, vedere l'esempio di IAM.

Esempio di utilizzo di pool di utenti Amazon Cognito:

```
{
  "id": "ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69",
  "payload": {
    "data": "{\"query\":\"subscription onCreateMessage {\n onCreateMessage {\n\n__typename\n message\n }\n }\",\"variables\":{\"}}\",
    \"extensions\": {
      \"authorization\": {
        \"Authorization\":
          \"eyJEXAMPLEiJjbG5xb3A5eW5MK09QYXIrMTJEXAMPLEBieU5WNHhsQjhPVW9YMnM2WldvPSIsImFsZyI6I1EXAMPLEEn0.e
qTCtrYeboUJ4luRSTPXaNewNeEXAMPLE14C6sfg05t00f0MpiUwj9k19gtNCCMqoSsjtQoUweFnH4JYa5EXAMPLEVx0yQEQ
RWvW7yQU3sNQRLEXAMPLEcd0yufBiCYs3dfQxTTdvR1B6Wz6CD781fNeKqfzzUn2beMoup2h6EXAMPLE4ow8cUPUPvG0DzR
        \"host\": \"example1234567890000.appsync-api.us-east-1.amazonaws.com\"
      }
    }
  }
}
```

```

    }
  }
},
"type": "start"
}

```

Esempio di utilizzo di IAM:

```

{
  "id": "eEXAMPLE-cf23-1234-5678-152EXAMPLE69",
  "payload": {
    "data": "{\"query\":\"subscription onCreateMessage {\n\n onCreateMessage {\n\n\n__typename\n\n message\n\n }\n\n }\",\"variables\":{\"}}\",
    \"extensions\": {
      \"authorization\": {
        \"accept\": \"application/json, text/javascript\",
        \"content-type\": \"application/json; charset=UTF-8\",
        \"X-Amz-Security-Token\":
        \"AgEXAMPLEZ2luX2VjEAoaDmFwLXNvdXR0ZWFEXAMPLEcwRQIgAh97Cljq7w0PL8KsxP3YtDuyC/9hAj8PhJ7Fvf38SgoC
+
+pEagWCveZUjKE0zyUhBEXAMPLEjj//////////8BEXAMPLEx0Dk2NDgyNzg1NSIMo1mWnpESWUoYw4BkKqEFSrm3DXuL8
+ZbVc4JKjDP4vUCKNR6Le9C9pZp9Psw0NoFy3vLBUDAXEXAMPLE0VG8feXfiEEA+1khgFK/
wEtWR+9zF7NaMMmSe07wN2gG2tH0eKMEXAMPLEEQX+sMbytQo8iepP9PZ0z1ZsSFb/
dP5Q8hk6YEXAMPLEYcKZsTkDAq2uKFQ8mYUVA9EtQnNRiFLEY83aKvG/tqLWNnG1SNVx7SMcfovkFDqQamm
+88y10wwAEYK7qcocex6Z7GGcaYuIfGpaX2MCCELeQvZ+8WxEg0nIfz7GYvsYNjLZSaRnV4G
+ILY1F0QNW64S9Nvj
+BwDg3ht2CrNvpwVY1j9U3nmxE0UG5ne83LL5hhqMpm25kmL7enVgw2kQzmU2id4IKu0C/
WaoDRu02F5zE63vJbxN8AYs7338+4B4HBB6BZ60Ugg96Q15RA41/
gIqxaVPxyTpDfTU5GfSLxocdYeniqqpFMtZG2n9d0u7GsQNcFkNcG3qDZm4tDo8tZbuym0a2VcF2E5hFEgXBa
+XLJCfXi/770qAEjP0x7Qdk3B43p8KG/BaioP5RsV8zBGvH1zAgyPha2rN70/
tT13yrmPd5QYEFwzexjKrV4mWIuRg8NTHYSZJUaeyCwTom80VFUJXG
+GYTUyv5W22aBcnoRGiCiKEYTL0kgXecdKFTHmcIAejQ9Welr0a196Kq87w5KNMckcCGFnwBNFLmfmbpNqT6rUBxss3X5nt
aox0FtHX21eF6qIGT8j1z+l2opU+ggwUgkhUUgCH2TfqBj+MLMVvpvgqJsPKt582caFKArIFIv0
+9QupxLnEH2hz04TMTfnU6bQC6z1buVe7h
+t0Lnh1YPFsLQ88anib/7TTC8k9DsBTq0ASE8R2GbSEsm09qbbMwgEaYUh0KtGeyQsSJdhSk6XxXThrWL9EnwBCXDkICMqd
+WgtPtK00weDlCaRs3R2qXcbNgVhleMk4IwnF8D1695AenU1LwHj0JLkCjxgNFfiWAFEPH9aEXAMPLExA==\",
    \"Authorization\": \"AWS4-HMAC-SHA256 Credential=XXXXXXXXXXXXXXXXXXXX/20200401/
us-east-1/appsyc/aws4_request, SignedHeaders=accept;content-
encoding;content-type;host;x-amz-date;x-amz-security-token,
Signature=b90131a61a7c4318e1c35ead5dbfdeb46339a7585bbdbceceaff51f4022eb1fd\",
    \"content-encoding\": \"amz-1.0\",
    \"host\": \"example1234567890000.appsyc-api.us-east-1.amazonaws.com\",
    \"x-amz-date\": \"20200401T001010Z\"
  }
}

```

```
    }
  }
},
"type": "start"
}
```

Esempio di utilizzo di un nome di dominio personalizzato:

```
{
  "id": "key-cf23-4cb8-9fcb-152ae4fd1e69",
  "payload": {
    "data": "{\"query\":\"subscription onCreateMessage {\\n  onCreateMessage {\\n    __typename\\n    message\\n  }\\n }\\n\", \"variables\":{}}",
    "extensions": {
      "authorization": {
        "x-api-key": "da2-12345678901234567890123456",
        "host": "api.example.com"
      }
    }
  },
  "type": "start"
}
```

Non è necessario aggiungere la firma SigV4 /connect all'URL e l'operazione GraphQL con stringa JSON la sostituisce. data Di seguito è riportato un esempio di richiesta di firma SigV4:

```
{
  url: "https://example1234567890000.appsync-api.us-east-1.amazonaws.com/graphql",
  data: "{\"query\":\"subscription onCreateMessage {\\n onCreateMessage {\\n __typename\\n message\\n }\\n }\\n}\",\"variables\":{\"}}\",
  method: "POST",
  headers: {
    "accept": "application/json, text/javascript",
    "content-encoding": "amz-1.0",
    "content-type": "application/json; charset=UTF-8",
  }
}
```

Messaggio di conferma dell'iscrizione

Dopo aver inviato il messaggio di avvio dell'abbonamento, il cliente deve AWS AppSync attendere l'invio del `start_ack` messaggio. Il `start_ack` messaggio indica che l'abbonamento è andato a buon fine.

Esempio di riconoscimento dell'abbonamento:

```
{
  "type": "start_ack",
  "id": "eEXAMPLE-cf23-1234-5678-152EXAMPLE69"
}
```

Messaggio di errore

Se l'avvio della connessione o la registrazione dell'abbonamento falliscono o se una sottoscrizione viene interrotta dal server, il server invia un messaggio di errore al client. Se l'errore si verifica durante l'avvio della connessione, la connessione verrà chiusa dal server.

- `"type": "error"`: un parametro `<string>` costante.
- `"id": <string>`: L'ID dell'abbonamento registrato corrispondente, se pertinente.
- `"payload" <Object>`: Un oggetto che contiene le informazioni di errore corrispondenti.

Esempio:

```
{
  "type": "error",
  "payload": {
    "errors": [
      {
        "errorType": "LimitExceededError",
        "message": "Rate limit exceeded"
      }
    ]
  }
}
```

Elaborazione dei messaggi dati

Quando un client invia una mutazione, AWS AppSync identifica tutti gli abbonati interessati e invia un `"type": "data"` messaggio a ciascuno di essi utilizzando l'abbonamento corrispondente `id` dall'operazione di sottoscrizione. `"start"` Il client dovrebbe tenere traccia dell'abbonamento inviato in modo `id` che, quando riceve un messaggio di dati, possa abbinarlo all'abbonamento corrispondente.

- `"type": "data"`: un parametro `<string>` costante.
- `"id": <string>`: L'ID dell'abbonamento registrato corrispondente.
- `"payload" <Object>`: Un oggetto che contiene le informazioni sulla sottoscrizione.

Esempio:

```
{
  "type": "data",
  "id": "ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69",
  "payload": {
    "data": {
      "onCreateMessage": {
        "__typename": "Message",
        "message": "test"
      }
    }
  }
}
```

Messaggio di annullamento registrazione sottoscrizione

Quando l'app desidera interrompere l'ascolto degli eventi di sottoscrizione, il client deve inviare un messaggio con il seguente oggetto JSON con stringhe:

- `"type": "stop"`: un parametro `<string>` costante.
- `"id": <string>`: L'ID dell'abbonamento di cui annullare la registrazione.

Esempio:

```
{
  "type": "stop",
```

```
"id": "ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69"
}
```

AWS AppSync restituisce un messaggio di conferma con il seguente oggetto JSON con stringhe:

- "type": "complete": un parametro <string> costante.
- "id": <string>: L'ID dell'abbonamento non registrato.

Dopo aver ricevuto il messaggio di conferma, il cliente non riceve più messaggi per questo particolare abbonamento.

Esempio:

```
{
  "type": "complete",
  "id": "eEXAMPLE-cf23-1234-5678-152EXAMPLE69"
}
```

Disconnettere il WebSocket

Prima della disconnessione, per evitare la perdita di dati, il client deve disporre della logica necessaria per verificare che al momento non sia in corso alcuna operazione tramite la WebSocket connessione. Tutti gli abbonamenti devono essere annullati prima di disconnettersi da WebSocket.

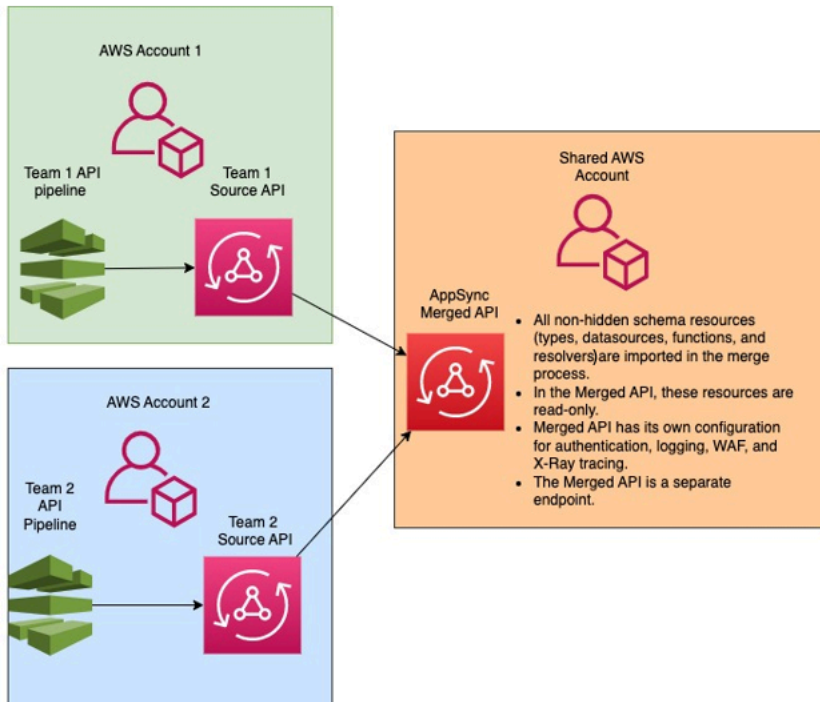
Fusione APIs AWS AppSync

Man mano che l'uso di GraphQL si espande all'interno di un'organizzazione, possono sorgere compromessi tra API ease-of-use e velocità di sviluppo delle API. Da un lato, le organizzazioni adottano AWS AppSync GraphQL per semplificare lo sviluppo delle applicazioni. Ciò offre agli sviluppatori un'API flessibile che possono utilizzare per accedere, manipolare e combinare in modo sicuro i dati di uno o più domini di dati con una singola chiamata di rete. D'altra parte, i team all'interno di un'organizzazione responsabili dei diversi domini di dati combinati in un unico endpoint dell'API GraphQL potrebbero desiderare la possibilità di creare, gestire e distribuire gli aggiornamenti delle API indipendentemente l'uno dall'altro. Ciò aumenta la loro velocità di sviluppo.

Per risolvere questa tensione, la APIs funzionalità AWS AppSync Merged consente ai team di diversi domini di dati di creare e implementare in modo indipendente AWS AppSync APIs (ad esempio schemi GraphQL, resolver, fonti di dati e funzioni), che possono quindi essere combinati in un'unica API unita. Ciò offre alle organizzazioni la possibilità di mantenere un'API multidominio semplice da

usare e un modo per i diversi team che contribuiscono a tale API di effettuare aggiornamenti delle API in modo rapido e indipendente.

Il diagramma seguente mostra il flusso di lavoro delle API unite:



Utilizzando Merged APIs, le organizzazioni possono importare le risorse di più fonti indipendenti AWS AppSync APIs in un unico endpoint API unito. AWS AppSync A tale scopo, AWS AppSync consente di creare un elenco di fonti e quindi unire tutti i metadati associati alla fonte AWS AppSync APIs, APIs inclusi schema, tipi, origini dati, resolver e funzioni, in una nuova API unita. AWS AppSync

Durante le unioni, esiste la possibilità che si verifichi un conflitto di unione a causa di incongruenze nel contenuto dei dati dell'API di origine, ad esempio conflitti di denominazione dei tipi quando si combinano più schemi. Per casi d'uso semplici in cui nessuna definizione nell'origine è in APIs conflitto, non è necessario modificare gli schemi dell'API di origine. L'API Merged risultante importa semplicemente tutti i tipi, i resolver, le fonti di dati e le funzioni dalla fonte originale. AWS AppSync APIs Nei casi d'uso complessi in cui sorgono conflitti, users/teams dovranno risolverli con vari mezzi. AWS AppSync fornisce agli utenti diversi strumenti ed esempi in grado di ridurre i conflitti di fusione.

Le fusioni successive configurate in AWS AppSync propagheranno le modifiche apportate nell'origine APIs all'API unita associata.

Unione e federazione APIs

Esistono molte soluzioni e modelli nella community GraphQL per combinare schemi GraphQL e consentire la collaborazione in team attraverso un grafico condiviso. AWS AppSync Merged APIs adotta un approccio alla composizione dello schema basato sulla fase di compilazione, in cui APIs sorgenti vengono combinati in un'API unita separata. Un approccio alternativo consiste nel sovrapporre un router in fase di esecuzione su più sorgenti APIs o sottografi. In questo approccio, il router riceve una richiesta, fa riferimento a uno schema combinato che mantiene come metadati, costruisce un piano di richiesta e quindi distribuisce gli elementi della richiesta tra i sottografi/server sottostanti. La tabella seguente confronta l'approccio in fase di compilazione dell'API AWS AppSync unita con gli approcci di run-time basati su router alla composizione dello schema GraphQL:

Funzionalità	AppSync API unita	Soluzioni basate su router
Sottografi gestiti in modo indipendente	Sì	Sì
Sottografi indirizzabili indipendentemente	Sì	Sì
Composizione automatica dello schema	Sì	Sì
Rilevamento automatico dei conflitti	Sì	Sì
Risoluzione dei conflitti tramite direttive dello schema	Sì	Sì
Server di sottografi supportati	AWS AppSync*	Può variare
Complessità della rete	Un'unica API unita significa nessun salto di rete aggiuntivo.	L'architettura a più livelli richiede la pianificazione e la delega delle query, l'analisi e la serializzazione/deserializzazione delle sottoquery e l'utilizzo di resolver di riferimen

		to in sottografi per eseguire i join.
Supporto all'osservabilità	Monitoraggio, registrazione e tracciamento integrati. Un unico server API integrato significa un debug semplificato.	Build-your-own osservabilità sul router e su tutti i server di sottografo associati. Debug complesso su sistemi distribuiti.
Supporto per l'autorizzazione	Supporto integrato per più modalità di autorizzazione.	Build-your-own regole di autorizzazione.
Sicurezza su più account	Supporto integrato per le associazioni di account tra AWS cloud.	Build-your-own modello di sicurezza.
Supporto per gli abbonamenti	Sì	No

* L' AWS AppSync unione APIs può essere associata AWS AppSync solo alla fonte. APIs Se hai bisogno di supporto per la composizione dello schema tra AWS AppSync e non AWS AppSync sotto grafici, puoi connettere uno o più GraphQL and/or Merged a una AWS AppSync soluzione basata su router. APIs [Ad esempio, consulta il blog di riferimento per l'aggiunta AWS AppSync APIs come sottografo utilizzando un'architettura basata su router con Apollo Federation v2: Apollo GraphQL Federation with. AWS AppSync](#)

Argomenti

- [Risoluzione unificata dei conflitti tramite API](#)
- [Configurazione degli schemi](#)
- [Configurazione delle modalità di autorizzazione](#)
- [Configurazione dei ruoli di esecuzione](#)
- [Configurazione della fusione tra account tramite APIs AWS RAM](#)
- [Unire](#)
- [Supporto aggiuntivo per Merged APIs](#)
- [Limitazioni delle API unite](#)
- [Considerazioni sull'API unita](#)

- [Creazione di un sistema unito APIs](#)

Risoluzione unificata dei conflitti tramite API

In caso di conflitto di fusione, AWS AppSync fornisce agli utenti diversi strumenti ed esempi per aiutare a risolvere il/i problema/i.

Direttive dello schema API unite

AWS AppSync ha introdotto diverse direttive GraphQL che possono essere utilizzate per ridurre o risolvere i conflitti tra i sorgenti: APIs

- **@canonical:** questa direttiva imposta la precedenza di tipi/campi con nomi e dati simili. Se due o più sorgenti APIs hanno lo stesso tipo o campo GraphQL, una di esse APIs può annotare il loro tipo o campo come canonico, a cui verrà data la priorità durante l'unione. Le informazioni in conflitto types/fields che non sono annotate con questa direttiva in un'altra fonte vengono ignorate quando vengono unite. APIs
- **@hidden:** questa direttiva incapsula alcuni elementi types/fields per rimuoverla dal processo di fusione. I team potrebbero voler rimuovere o nascondere tipi o operazioni specifici nell'API di origine in modo che solo i client interni possano accedere a dati digitati specifici. Con questa direttiva allegata, i tipi o i campi non vengono uniti nell'API Merged.
- **@renamed:** questa direttiva modifica i nomi di types/fields per ridurre i conflitti di denominazione. Ci sono situazioni in cui diversi APIs hanno lo stesso tipo o nome di campo. Tuttavia, devono essere tutti disponibili nello schema unito. Un modo semplice per includerli tutti nell'API Merged consiste nel rinominare il campo con qualcosa di simile ma diverso.

Per mostrare lo schema di utilità fornito dalle direttive, considera il seguente esempio:

In questo esempio, supponiamo di voler unire due sorgenti. APIs Ci vengono forniti due schemi per creare e recuperare i post (ad esempio, sezione commenti o post sui social media). Supponendo che i tipi e i campi siano molto simili, c'è un'alta probabilità di conflitto durante un'operazione di unione. I frammenti seguenti mostrano i tipi e i campi di ogni schema.

Il primo file, chiamato Source1.graphQL, è uno schema GraphQL che consente a un utente di creare utilizzando la mutazione. Posts putPost Ciascuno Post contiene un titolo e un ID. L'ID viene utilizzato per fare riferimento alle informazioni del poster (e-mail e indirizzo) e alMessage, o al payload (contenuto). User Il User tipo è annotato con il tag @canonical.

```
# This snippet represents a file called Source1.graphql

type Mutation {
  putPost(id: ID!, title: String!): Post
}

type Post {
  id: ID!
  title: String!
}

type Message {
  id: ID!
  content: String
}

type User @canonical {
  id: ID!
  email: String!
  address: String!
}

type Query {
  singlePost(id: ID!): Post
  getMessage(id: ID!): Message
}
```

Il secondo file, chiamato `Source2.graphQL`, è uno schema GraphQL che fa cose molto simili a `Source1.graphQL`. Tuttavia, notate che i campi di ogni tipo sono diversi. Quando si uniscono questi due schemi, si verificheranno conflitti di unione dovuti a queste differenze.

Nota anche come `Source2.graphQL` contenga anche diverse direttive per ridurre questi conflitti. Il `Post` tipo è annotato con un tag `@hidden` per offuscarsi durante l'operazione di unione. Il `Message` tipo è annotato con il tag `@renamed` per modificare il nome del tipo `ChatMessage` in caso di conflitto di denominazione con un altro tipo. `Message`

```
# This snippet represents a file called Source2.graphql

type Post @hidden {
  id: ID!
  title: String!
  internalSecret: String!
}
```

```

}

type Message @renamed(to: "ChatMessage") {
  id: ID!
  chatId: ID!
  from: User!
  to: User!
}

# Stub user so that we can link the canonical definition from Source1
type User {
  id: ID!
}

type Query {
  getPost(id: ID!): Post
  getMessage(id: ID!): Message @renamed(to: "getChatMessage")
}

```

Quando si verifica l'unione, il risultato produrrà il file: `MergedSchema.graphql`

```

# This snippet represents a file called MergedSchema.graphql

type Mutation {
  putPost(id: ID!, title: String!): Post
}

# Post from Source2 was hidden so only uses the Source1 definition.
type Post {
  id: ID!
  title: String!
}

# Renamed from Message to resolve the conflict
type ChatMessage {
  id: ID!
  chatId: ID!
  from: User!
  to: User!
}

type Message {
  id: ID!
}

```

```
    content: String
  }

# Canonical definition from Source1
type User {
  id: ID!
  email: String!
  address: String!
}

type Query {
  singlePost(id: ID!): Post
  getMessage(id: ID!): Message

  # Renamed from getMessage
  getChatMessage(id: ID!): ChatMessage
}
```

Durante l'unione si sono verificate diverse cose:

- Il `User` tipo di `Source1.graphQL` ha avuto la priorità rispetto a quello di `Source2.graphQL` grazie all'annotazione `User @canonical`.
- Il `Message` tipo di `Source1.graphQL` è stato incluso nell'unione. Tuttavia, il file di `Source2.graphQL` presentava un `Message` conflitto di denominazione. Grazie alla sua annotazione `@renamed`, è stata inclusa anche nell'unione ma con il nome alternativo. `ChatMessage`
- Il `Post` tipo di `Source1.graphQL` è stato incluso, ma non il tipo di `Source2.graphQL`. `Post` Normalmente, si verificava un conflitto su questo tipo, ma poiché il `Post` tipo di `Source2.GraphQL` aveva un'annotazione `@hidden`, i suoi dati erano offuscati e non inclusi nell'unione. Ciò non ha provocato conflitti.
- Il `Query` tipo è stato aggiornato per includere il contenuto di entrambi i file. Tuttavia, una `GetMessage` query è stata rinominata in `GetChatMessage` base alla direttiva. Ciò ha risolto il conflitto di denominazione tra le due query con lo stesso nome.

C'è anche il caso in cui nessuna direttiva venga aggiunta a un tipo in conflitto. Qui, il tipo unito includerà l'unione di tutti i campi di tutte le definizioni di origine di quel tipo. Ad esempio, si consideri il seguente esempio:

Questo schema, chiamato `Source1.graphQL`, consente la creazione e il recupero. `Posts` La configurazione è simile all'esempio precedente, ma con meno informazioni.

```
# This snippet represents a file called Source1.graphql

type Mutation {
  putPost(id: ID!, title: String!): Post
}

type Post {
  id: ID!
  title: String!
}

type Query {
  getPost(id: ID!): Post
}
```

Questo schema, chiamato Source2.GraphQL, consente di creare e recuperare Reviews (ad esempio, la valutazione dei film o le recensioni dei ristoranti). Review sono associati allo stesso valore ID. Post. Insieme, contengono il titolo, l'ID del post e il messaggio di payload del post completo della recensione.

Durante la fusione, si verificherà un conflitto tra i due Post tipi. Poiché non esistono annotazioni per risolvere questo problema, il comportamento predefinito consiste nell'eseguire un'operazione di unione sui tipi in conflitto.

```
# This snippet represents a file called Source2.graphql

type Mutation {
  putReview(id: ID!, postId: ID!, comment: String!): Review
}

type Post {
  id: ID!
  reviews: [Review]
}

type Review {
  id: ID!
  postId: ID!
  comment: String!
}

type Query {
```

```
  getReview(id: ID!): Review
}
```

Quando si verifica l'unione, il risultato produrrà il file: `MergedSchema.graphql`

```
# This snippet represents a file called MergedSchema.graphql

type Mutation {
  putReview(id: ID!, postId: ID!, comment: String!): Review
  putPost(id: ID!, title: String!): Post
}

type Post {
  id: ID!
  title: String!
  reviews: [Review]
}

type Review {
  id: ID!
  postId: ID!
  comment: String!
}

type Query {
  getPost(id: ID!): Post
  getReview(id: ID!): Review
}
```

Durante l'unione si sono verificate diverse cose:

- Il `Mutation` tipo non ha avuto conflitti ed è stato unito.
- I campi `Post` tipo sono stati combinati tramite un'operazione di unione. Notate come l'unione tra i due abbia prodotto un singolo `id` `title`, un `e` un singolo `reviews`.
- Il `Review` tipo non ha subito conflitti ed è stato unito.
- Il `Query` tipo non ha riscontrato conflitti ed è stato unito.

Gestione dei resolver su tipi condivisi

Nell'esempio precedente, considera il caso in cui Source1.GraphQL ha configurato un resolver di unità su, `Query.getPost` che utilizza un'origine dati DynamoDB denominata. `PostDatasource`. Questo resolver restituirà un tipo e. `id title Post`. Consideriamo ora che Source2.graphQL ha configurato un resolver di pipeline su, che esegue due funzioni. `Post.reviews`. `Function1` ha una fonte di `None` dati allegata per eseguire controlli di autorizzazione personalizzati. `Function2` ha un'origine dati DynamoDB allegata per interrogare la tabella. `reviews`

```
query GetPostQuery {
  getPost(id: "1") {
    id,
    title,
    reviews
  }
}
```

Quando la query precedente viene eseguita da un client all'endpoint Merged API, il AWS AppSync servizio esegue innanzitutto il resolver di unità per `Query.getPost` from `Source1`, che chiama `PostDatasource` e restituisce i dati da DynamoDB. Quindi, esegue il resolver della `Post.reviews` pipeline in cui `Function1` esegue una logica di autorizzazione personalizzata e restituisce le recensioni fornite. `Function2` id `$context.source`. Il servizio elabora la richiesta come una singola esecuzione GraphQL e questa semplice richiesta richiederà solo un singolo token di richiesta.

Gestione dei conflitti dei resolver su tipi condivisi

Considera il seguente caso in cui implementiamo anche un resolver on per fornire più campi contemporaneamente oltre al resolver di campo `Query.getPost` in cui è inserito. `Source2`. `Source1.graphQL` può assomigliare a questo:

```
# This snippet represents a file called Source1.graphql

type Post {
  id: ID!
  title: String!
  date: AWSDateTime!
}

type Query {
  getPost(id: ID!): Post
```



```
}
```

Source2.graphQL può assomigliare a questo:

```
# This snippet represents a file called Source2.graphql

type Post {
  id: ID!
  content: String!
  contentHash: String!
  author: String!
}

type Query {
  getPost(id: ID!): Post
}
```

Il tentativo di unire questi due schemi genererà un errore di unione perché Merged APIs non consente di collegare più AWS AppSync resolver di origine allo stesso campo. Per risolvere questo conflitto, puoi implementare un pattern di risoluzione dei campi che richieda a Source2.GraphQL di aggiungere un tipo separato che definirà i campi di sua proprietà rispetto al tipo. `Post`. Nell'esempio seguente, aggiungiamo un tipo chiamato `PostInfo`, che contiene i campi `content` e `author` che verranno risolti da Source2.graphQL. Source1.graphQL implementerà il resolver collegato a `Query.getPost`, mentre Source2.graphQL ora collegherà un resolver per garantire che tutti i dati possano essere recuperati con successo: `Post.postInfo`

```
type Post {
  id: ID!
  postInfo: PostInfo
}

type PostInfo {
  content: String!
  contentHash: String!
  author: String!
}

type Query {
  getPost(id: ID!): Post
}
```

Sebbene la risoluzione di tale conflitto richieda la riscrittura degli schemi delle API di origine e, potenzialmente, la modifica delle query da parte dei clienti, il vantaggio di questo approccio è che la proprietà dei resolver uniti rimane chiara tra i team di origine.

Configurazione degli schemi

Due parti sono responsabili della configurazione degli schemi per la creazione di un'API unita:

- Proprietari delle API unite: i proprietari delle API unite devono configurare la logica di autorizzazione dell'API unita e le impostazioni avanzate come la registrazione, il tracciamento, la memorizzazione nella cache e il supporto WAF.
- Proprietari delle API di origine associate: i proprietari delle API associate devono configurare gli schemi, i resolver e le origini dati che compongono l'API unita.

Poiché lo schema dell'API Merged viene creato dagli schemi della fonte associata, è di sola lettura. Ciò significa che le modifiche allo schema devono essere avviate nella fonte. Nella AWS AppSync console, puoi passare dallo schema Merged ai singoli schemi della fonte API inclusi nell'API Merged utilizzando l'elenco a discesa sopra la finestra Schema.

Configurazione delle modalità di autorizzazione

Sono disponibili diverse modalità di autorizzazione per proteggere l'API unita. Per ulteriori informazioni sulle modalità di autorizzazione in AWS AppSync, consulta [Autorizzazione e autenticazione](#).

Le seguenti modalità di autorizzazione sono disponibili per l'uso con Merged: APIs

- Chiave API: la strategia di autorizzazione più semplice. Tutte le richieste devono includere una chiave API sotto l'intestazione della `x-api-key` richiesta. Le chiavi API scadute vengono conservate per 60 giorni dopo la data di scadenza.
- AWS Identity and Access Management (IAM): AWS la strategia di autorizzazione IAM autorizza tutte le richieste firmate con sigv4.
- Pool di utenti Amazon Cognito: autorizza i tuoi utenti tramite i pool di utenti di Amazon Cognito per ottenere un controllo più preciso.
- AWS Autorizzatori Lambda: una funzione serverless che consente di autenticare e autorizzare l'accesso all'API utilizzando una logica personalizzata. AWS AppSync

- OpenID Connect: questo tipo di autorizzazione applica i token OpenID connect (OIDC) forniti da un servizio conforme a OIDC. La tua applicazione può usare gli utenti e i privilegi definiti dal provider OIDC per controllare l'accesso.

Le modalità di autorizzazione di un'API unita sono configurate dal proprietario dell'API unita. Al momento di un'operazione di unione, l'API unita deve includere la modalità di autorizzazione principale configurata su un'API di origine come modalità di autorizzazione principale propria o come modalità di autorizzazione secondaria. In caso contrario, sarà incompatibile e l'operazione di unione avrà esito negativo a causa di un conflitto. Quando si utilizzano direttive multi-auth nell'origine APIs, il processo di fusione è in grado di unire automaticamente queste direttive nell'endpoint unificato. Nel caso in cui la modalità di autorizzazione primaria dell'API di origine non corrisponda alla modalità di autorizzazione principale dell'API unita, aggiungerà automaticamente queste direttive di autenticazione per garantire che la modalità di autorizzazione per i tipi nell'API di origine sia coerente.

Configurazione dei ruoli di esecuzione

Quando si crea un'API unita, è necessario definire un ruolo di servizio. Un ruolo AWS di servizio è un ruolo di AWS Identity and Access Management (IAM) utilizzato AWS dai servizi per eseguire attività per conto dell'utente.

In questo contesto, è necessario che l'API Merged esegua resolver che accedano ai dati dalle fonti di dati configurate nell'origine. APIs Il ruolo di servizio richiesto a tal fine è `mergedApiExecutionRole`, e deve disporre dell'accesso esplicito all'esecuzione delle richieste sull'origine APIs inclusa nell'API unita tramite l'autorizzazione IAM. `appsync:SourceGraphQL` Durante l'esecuzione di una richiesta GraphQL, il AWS AppSync servizio assumerà questo ruolo di servizio e autorizzerà il ruolo a eseguire l'azione. `appsync:SourceGraphQL`

AWS AppSync supporta l'autorizzazione o la negazione di questa autorizzazione su specifici campi di primo livello all'interno della richiesta, ad esempio il funzionamento della modalità di autorizzazione IAM per IAM. APIs Per non-top-level i campi, AWS AppSync richiede di definire l'autorizzazione sull'ARN dell'API di origine stesso. Per limitare l'accesso a non-top-level campi specifici nell'API Merged, ti consigliamo di implementare una logica personalizzata all'interno di Lambda o di nascondere i campi dell'API di origine all'API Merged utilizzando la direttiva `@hidden`. Se desideri consentire al ruolo di eseguire tutte le operazioni sui dati all'interno di un'API di origine, puoi aggiungere la politica seguente. Tieni presente che la prima voce di risorsa consente l'accesso a tutti i campi di primo livello e la seconda voce riguarda i resolver secondari che autorizzano la risorsa API di origine stessa:

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Action": [ "appsync:SourceGraphQL" ],
    "Resource": [
      "arn:aws:appsync:us-west-2:123456789012:apis/YourSourceGraphQLApiId/*",
      "arn:aws:appsync:us-west-2:123456789012:apis/YourSourceGraphQLApiId" ]
  }]
}
```

Se desideri limitare l'accesso solo a uno specifico campo di primo livello, puoi utilizzare una politica come questa:

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Action": [ "appsync:SourceGraphQL" ],
    "Resource": [
      "arn:aws:appsync:us-west-2:123456789012:apis/YourSourceGraphQLApiId/types/Query/fields/<Field-1>",
      "arn:aws:appsync:us-west-2:123456789012:apis/YourSourceGraphQLApiId" ]
  }]
}
```

Puoi anche utilizzare la procedura guidata di creazione dell'API della AWS AppSync console per generare un ruolo di servizio che consenta all'API unita di accedere alle risorse configurate nel codice sorgente APIs che si trovano nello stesso account dell'API unita. Nel caso in cui la fonte non si APIs trovi nello stesso account dell'API unita, devi prima condividere le tue AWS risorse utilizzando Resource Access Manager (AWS RAM).

Configurazione della fusione tra account tramite APIs AWS RAM

Quando crei un'API unita, puoi facoltativamente associare la fonte APIs da altri account che sono stati condivisi tramite AWS Resource Access Manager (AWS RAM). AWS RAM ti aiuta a condividere le tue risorse in modo sicuro tra AWS gli account, all'interno dell'organizzazione o delle unità organizzative (OUs) e con i ruoli e gli utenti IAM.

AWS AppSync si integra con AWS RAM per supportare la configurazione e l'accesso ai sorgenti APIs su più account da un'unica API unita. AWS RAM consente di creare una condivisione di risorse o un contenitore di risorse e i set di autorizzazioni che verranno condivisi per ciascuna di esse. È possibile aggiungere AWS AppSync APIs a una condivisione di risorse in AWS RAM. All'interno di una condivisione di risorse, AWS AppSync fornisce tre diversi set di autorizzazioni che possono essere associati a un' AWS AppSync API nella RAM:

1. `AWSRAMPermissionAppSyncSourceApiOperationAccess`: il set di autorizzazioni predefinito che viene aggiunto quando si condivide un' AWS AppSync API AWS RAM se non viene specificata nessun'altra autorizzazione. Questo set di autorizzazioni viene utilizzato per condividere un' AWS AppSync API di origine con un proprietario di un'API unita. Questo set di autorizzazioni include l'autorizzazione per `appsync:AssociateMergedGraphQLApi` l'API di origine e l'`appsync:SourceGraphQL` autorizzazione richiesta per accedere alle risorse dell'API di origine in fase di esecuzione.
2. `AWSRAMPermissionAppSyncMergedApiOperationAccess`: questo set di autorizzazioni deve essere configurato quando si condivide un'API unita con un proprietario dell'API di origine. Questo set di autorizzazioni darà all'API di origine la possibilità di configurare l'API unita, inclusa la possibilità di associare qualsiasi origine di APIs proprietà del principale di destinazione all'API unita e di leggere e aggiornare le associazioni delle API di origine dell'API unita.
3. `AWSRAMPermissionAppSyncAllowSourceGraphQLAccess`: Questo set di autorizzazioni consente di utilizzare l'`appsync:SourceGraphQL` autorizzazione con un'API. AWS AppSync È destinato a essere utilizzato per condividere un'API di origine con un proprietario di un'API unita. A differenza del set di autorizzazioni predefinito per l'accesso alle operazioni dell'API di origine, questo set di autorizzazioni include solo l'autorizzazione `appsync:SourceGraphQL` di runtime. Se un utente sceglie di condividere l'accesso all'operazione Merged API con un proprietario dell'API di origine, dovrà inoltre condividere questa autorizzazione dall'API di origine al proprietario dell'API unita per avere accesso al runtime tramite l'endpoint dell'API unita.

AWS AppSync supporta anche le autorizzazioni gestite dal cliente. Quando una delle autorizzazioni AWS gestite fornite non funziona, puoi creare un'autorizzazione gestita dal cliente personalizzata. Le

autorizzazioni gestite dal cliente sono autorizzazioni gestite che puoi creare e gestire specificando con precisione quali azioni possono essere eseguite in quali condizioni con risorse condivise. AWS RAM AWS AppSync consente di scegliere tra le seguenti azioni durante la creazione delle proprie autorizzazioni:

1. `appsync:AssociateSourceGraphQLApi`
2. `appsync:AssociateMergedGraphQLApi`
3. `appsync:GetSourceApiAssociation`
4. `appsync:UpdateSourceApiAssociation`
5. `appsync:StartSchemaMerge`
6. `appsync:ListTypesByAssociation`
7. `appsync:SourceGraphQL`

Dopo aver condiviso correttamente un'API di origine o un'API unita AWS RAM e, se necessario, l'invito alla condivisione delle risorse è stato accettato, l'invito alla condivisione delle risorse sarà visibile nella AWS AppSync console quando si creano o si aggiornano le associazioni delle API di origine sulla Merged API. Puoi anche elencare tutto ciò AWS AppSync APIs che è stato condiviso AWS RAM con il tuo account indipendentemente dall'autorizzazione impostata richiamando l'`ListGraphQLApis` operazione fornita AWS AppSync e utilizzando il filtro del `OTHER_ACCOUNTS` proprietario.

Note

La condivisione tramite AWS RAM richiede che il chiamante sia autorizzato a eseguire l'`appsync:PutResourcePolicy` azione su qualsiasi API condivisa. AWS RAM

Unire

Gestione delle unioni

APIs Le fusioni hanno lo scopo di supportare la collaborazione in team su un endpoint AWS AppSync unificato. I team possono far evolvere in modo indipendente la propria fonte isolata APIs GraphQL nel backend, mentre AWS AppSync il servizio gestisce l'integrazione delle risorse nel singolo endpoint Merged API al fine di ridurre l'attrito nella collaborazione e ridurre i tempi di sviluppo.

Unioni automatiche

L'origine APIs associata alla tua API AWS AppSync unita può essere configurata per l'unione automatica (unione automatica) nell'API unita dopo aver apportato modifiche all'API di origine. Ciò garantisce che le modifiche dall'API di origine vengano sempre propagate all'endpoint dell'API unita in background. Qualsiasi modifica nello schema dell'API di origine verrà aggiornata nell'API unita a condizione che non introduca un conflitto di fusione con una definizione esistente nell'API unita. Se l'aggiornamento nell'API di origine sta aggiornando un resolver, un'origine dati o una funzione, verrà aggiornata anche la risorsa importata. Quando viene introdotto un nuovo conflitto che non può essere risolto automaticamente (risolto automaticamente), l'aggiornamento dello schema dell'API unita viene rifiutato a causa di un conflitto non supportato durante l'operazione di unione. Il messaggio di errore è disponibile nella console per ogni associazione di API di origine con lo stato di `MERGE_FAILED`. Puoi anche controllare il messaggio di errore chiamando l'GetSourceApiAssociationoperazione per una determinata associazione API di origine utilizzando l' AWS SDK o utilizzando la AWS CLI in questo modo:

```
aws appsync get-source-api-association --merged-api-identifier <Merged API ARN> --
association-id <SourceApiAssociation id>
```

Ciò produrrà un risultato nel seguente formato:

```
{
  "sourceApiAssociation": {
    "associationId": "<association id>",
    "associationArn": "<association arn>",
    "sourceApiId": "<source api id>",
    "sourceApiArn": "<source api arn>",
    "mergedApiArn": "<merged api arn>",
    "mergedApiId": "<merged api id>",
    "sourceApiAssociationConfig": {
      "mergeType": "MANUAL_MERGE"
    },
    "sourceApiAssociationStatus": "MERGE_FAILED",
    "sourceApiAssociationStatusDetail": "Unable to resolve conflict on object with
name title: Merging is not supported for fields with different types."
  }
}
```

Unioni manuali

L'impostazione predefinita per un'API di origine è un'unione manuale. Per unire le modifiche apportate all'origine APIs dall'ultimo aggiornamento dell'API unita, il proprietario dell'API di origine può richiamare un'unione manuale dalla AWS AppSync console o tramite l'`StartSchemaMerge` operazione disponibile nell'SDK AWS e nella CLI. AWS

Supporto aggiuntivo per Merged APIs

Configurazione degli abbonamenti

A differenza degli approcci basati su router alla composizione dello schema GraphQL, AWS AppSync Merged fornisce supporto APIs integrato per gli abbonamenti GraphQL. Tutte le operazioni di sottoscrizione definite nella fonte associata verranno automaticamente unite e APIs funzioneranno nell'API Merged senza modifiche. [Per ulteriori informazioni su come AWS AppSync supporta gli abbonamenti tramite WebSockets connessione serverless, consulta Dati in tempo reale.](#)

Configurazione dell'osservabilità

AWS AppSync [Merged APIs fornisce registrazione, monitoraggio e metriche integrate tramite Amazon. CloudWatch](#) AWS AppSync fornisce anche supporto integrato per il tracciamento tramite [AWS X-Ray](#)

Configurazione di domini personalizzati

AWS AppSync [Merged APIs fornisce supporto integrato per l'utilizzo di domini personalizzati con gli endpoint GraphQL e Real-time dell'API Merged.](#)

Configurazione della memorizzazione nella cache

AWS AppSync Merged APIs fornisce supporto integrato per la memorizzazione opzionale nella cache delle risposte a livello di richiesta e/o a livello di resolver, nonché per la compressione delle risposte. [Per ulteriori informazioni, consulta Caching e compressione.](#)

Configurazione privata APIs

AWS AppSync [Merged APIs fornisce un supporto integrato per Private APIs che limita l'accesso agli endpoint GraphQL e Real-time dell'API Merged al traffico proveniente dagli endpoint VPC che puoi configurare.](#)

Configurazione delle regole del firewall

AWS AppSync Merged APIs fornisce un supporto integrato per AWS WAF, che consente di proteggere le [applicazioni Web APIs definendo regole di firewall](#).

Configurazione dei log di controllo

AWS AppSync Merged APIs fornisce supporto integrato per AWS CloudTrail, che consente di [configurare e gestire](#) i registri di controllo.

Limitazioni delle API unite

Durante lo sviluppo di Merged APIs, prendi nota delle seguenti regole:

1. Un'API unita non può essere un'API di origine per un'altra API unita.
2. Un'API di origine non può essere associata a più di un'API unita.
3. Il limite di dimensione predefinito per un documento dello schema dell'API unita è di 10 MB.
4. Il numero predefinito di sorgenti APIs che può essere associato a un'API unita è 10. Tuttavia, puoi richiedere un aumento del limite se hai bisogno di più di 10 sorgenti APIs nella tua API unita.

Considerazioni sull'API unita

Durante la progettazione e l'implementazione di Merged APIs, tenete presente quanto segue:

L'unione di più sorgenti APIs in un unico endpoint può aumentare le dimensioni e la complessità dello schema e delle query GraphQL. Man mano che lo schema unito cresce, potrebbe essere necessario che le query attraversino più resolver per soddisfare una singola richiesta, il che può aggiungere latenza al tempo complessivo della richiesta. Ad esempio, una query che accede ai campi da più fonti APIs può richiedere l'esecuzione di resolver da ciascuna API di origine in sequenza, con ogni resolver che si aggiunge AWS AppSync al tempo di risposta totale.

Ti consigliamo vivamente di testare APIs accuratamente i dispositivi Merged durante lo sviluppo e in condizioni di carico realistiche per assicurarti che soddisfino i requisiti aziendali. Presta particolare attenzione a:

- La profondità e la complessità dello schema unito, in particolare delle query che accedono ai campi di più fonti. APIs
- Il numero di resolver che devono essere eseguiti per soddisfare i modelli di query comuni.
- Le caratteristiche prestazionali delle sorgenti di dati e dei resolver in base al carico previsto.

- L'impatto della latenza di rete sull'accesso alle risorse da più fonti. APIs

Prendi in considerazione l'implementazione di ottimizzazioni delle prestazioni come la memorizzazione nella cache, il raggruppamento in batch delle richieste di origine dati e la progettazione di schemi API di origine per ridurre al minimo il numero di esecuzioni di resolver necessarie per le operazioni comuni.

Creazione di un sistema unito APIs

Per creare un'API unita nella console

1. Accedi Console di gestione AWS e apri la [AWS AppSync console](#).
 - Nella dashboard, scegli Crea API.
 2. Scegli Merged API, quindi scegli Avanti.
 3. Nella pagina Specificare i dettagli dell'API, inserisci le seguenti informazioni:
 - a. In Dettagli API, inserisci le seguenti informazioni:
 - i. Specificate il nome dell'API unita. Questo campo consente di etichettare l'API GraphQL per distinguerla facilmente dagli altri GraphQL. APIs
 - ii. Specificate i dati di contatto. Questo campo è facoltativo e assegna un nome o un gruppo all'API GraphQL. Non è collegato o generato da altre risorse e funziona in modo molto simile al campo del nome dell'API.
 - b. In Service role, devi assegnare un ruolo di esecuzione IAM all'API unita in modo che AWS AppSync possa importare e utilizzare le tue risorse in modo sicuro in fase di esecuzione. Puoi scegliere di creare e utilizzare un nuovo ruolo di servizio, che ti consentirà di specificare le politiche e le risorse da utilizzare. AWS AppSync Puoi anche importare un ruolo IAM esistente selezionando Usa un ruolo di servizio esistente, quindi selezionando il ruolo dall'elenco a discesa.
 - c. In Configurazione API privata, puoi scegliere di abilitare le funzionalità dell'API privata. Tieni presente che questa scelta non può essere modificata dopo aver creato l'API unita. Per ulteriori informazioni su private APIs, consulta [Using AWS AppSync Private APIs](#).
- Dopo aver finito, scegli Avanti.
4. Successivamente, è necessario aggiungere il GraphQL APIs che verrà utilizzato come base per l'API unita. Nella APIs pagina Seleziona fonte, inserisci le seguenti informazioni:

- a. Nella tabella APIs dal tuo AWS account, scegli Aggiungi fonte APIs. Nell'elenco di GraphQL APIs, ogni voce conterrà i seguenti dati:
 - i. Nome: il campo del nome dell'API GraphQL.
 - ii. ID API: il valore ID univoco dell'API GraphQL.
 - iii. Modalità di autenticazione primaria: la modalità di autorizzazione predefinita per l'API GraphQL. Per ulteriori informazioni sulle modalità di autorizzazione in AWS AppSync, vedere [Autorizzazione e autenticazione](#).
 - iv. Modalità di autenticazione aggiuntiva: le modalità di autorizzazione secondarie configurate nell'API GraphQL.
 - v. Scegli quella APIs che utilizzerai nell'API unita selezionando la casella di controllo accanto al campo Nome dell'API. Successivamente, scegli Aggiungi fonte. APIs Il GraphQL selezionato APIs verrà visualizzato nella tabella APIs dei tuoi AWS conti.
 - b. Nella tabella APIs Da altri AWS conti, scegli Aggiungi sorgente APIs. Il GraphQL APIs in questo elenco proviene da altri account che condividono le proprie risorse con l'utente tramite AWS Resource Access Manager (AWS RAM). Il processo di selezione di GraphQL APIs in questa tabella è lo stesso del processo descritto nella sezione precedente. Per ulteriori informazioni sulla condivisione di risorse tramite AWS RAM, consulta [What is AWS Resource Access Manager?](#).
- Scegli Avanti dopo aver finito.
- c. Aggiungi la tua modalità di autenticazione principale. Vedi [Autorizzazione e autenticazione](#) per ulteriori informazioni. Scegli Next (Successivo).
 - d. Controlla i tuoi input, quindi scegli Crea API.

Creazione di GraphQL APIs con l'introspezione RDS

AWS AppSync l'utilità di introspezione può scoprire modelli dalle tabelle del database e proporre tipi GraphQL. La procedura guidata Create API della AWS AppSync console può generare istantaneamente un'API da un database Aurora MySQL o PostgreSQL. Crea automaticamente tipi e resolver per leggere e scrivere dati JavaScript.

AWS AppSync fornisce l'integrazione diretta con i database Amazon Aurora tramite l'API Amazon RDS Data. Aniché richiedere una connessione persistente al database, l'API Amazon RDS Data offre un endpoint HTTP sicuro a cui AWS AppSync connettersi per l'esecuzione SQL di

istruzioni. Puoi usarlo per creare un'API di database relazionale per i tuoi carichi di lavoro MySQL e PostgreSQL su Aurora.

La creazione di un'API per il database relazionale presenta diversi vantaggi: AWS AppSync

- Il database non è esposto direttamente ai client, il che significa che il punto di accesso è separato dal database stesso.
- È possibile creare API soluzioni personalizzate in base alle esigenze di diverse applicazioni, eliminando la necessità di utilizzare logiche aziendali personalizzate nei frontend. Questo è in linea con lo schema (BFF). Backend-For-Frontend
- L'autorizzazione e il controllo degli accessi possono essere implementati a AWS AppSync livello utilizzando varie modalità di autorizzazione per controllare l'accesso. Non sono necessarie risorse di elaborazione aggiuntive per connettersi al database, ad esempio l'hosting di un server Web o l'invio di connessioni tramite proxy.
- È possibile aggiungere funzionalità in tempo reale tramite abbonamenti, con le mutazioni dei dati effettuate AppSync automaticamente tramite push ai client connessi.
- I client possono connettersi all'API tramite HTTPS utilizzando porte comuni come 443.

AWS AppSync semplifica la creazione API a partire da database relazionali esistenti. La sua utilità di introspezione può scoprire modelli dalle tabelle del database e proporre tipi GraphQL. La procedura guidata Create API della AWS AppSync console può generare istantaneamente un'API da un database Aurora MySQL o PostgreSQL. Crea automaticamente tipi e resolver per leggere e scrivere dati JavaScript.

AWS AppSync fornisce JavaScript utilità integrate per semplificare la scrittura di istruzioni SQL nei resolver. È possibile utilizzare i modelli AWS AppSync di `sql` tag per istruzioni statiche con valori dinamici o le utilità del `rds` modulo per creare istruzioni a livello di codice. [Per ulteriori informazioni, consulta il riferimento alla funzione resolver per le fonti di dati RDS e i moduli integrati.](#)

Utilizzo della funzione di introspezione (console)

Per un tutorial dettagliato e una guida introduttiva, vedi [Tutorial: Aurora PostgreSQL Serverless with Data API](#).

La AWS AppSync console consente di creare un'API AWS AppSync GraphQL dal database Aurora esistente configurato con l'API Data in pochi minuti. Questo genera rapidamente uno schema operativo basato sulla configurazione del database. Puoi utilizzare l'API così com'è o basarti su di essa per aggiungere funzionalità.

1. [Accedi a Console di gestione AWS e apri la AppSync console.](#)

- Nel pannello di controllo, scegliere Create API (Crea API).

2. In Opzioni API, scegli GraphQL APIs, Inizia con un cluster Amazon Aurora, quindi Avanti.

- Inserisci un nome API. Verrà utilizzato come identificatore per l'API nella console.
- Per i dettagli di contatto, puoi inserire un punto di contatto per identificare un gestore dell'API. Questo campo è opzionale.
- In Configurazione API privata, puoi abilitare le funzionalità dell'API privata. È possibile accedere a un'API privata solo da un endpoint VPC configurato (VPCE). [Per ulteriori informazioni, consulta Privato. APIs](#)

Non è consigliabile abilitare questa funzionalità per questo esempio. Scegli Avanti dopo aver esaminato i dati inseriti.

3. Nella pagina Database, scegli Seleziona database.

- È necessario scegliere il database dal cluster. Il primo passo è scegliere la regione in cui si trova il cluster.
- Scegli il cluster Aurora dall'elenco a discesa. Tieni presente che devi aver creato e [abilitato](#) un'API di dati corrispondente prima di utilizzare la risorsa.
- Successivamente, è necessario aggiungere le credenziali del database al servizio. Questo viene fatto principalmente utilizzando Gestione dei segreti AWS. Scegli la regione in cui esiste il tuo segreto. Per ulteriori informazioni su come recuperare informazioni segrete, consulta [Trova segreti](#) o [Recupera](#) segreti.
- Aggiungi il tuo segreto dall'elenco a discesa. Tieni presente che l'utente deve disporre delle [autorizzazioni di lettura](#) per il tuo database.

4. Scegli Importa.

AWS AppSync inizierà a esaminare il database, scoprendo tabelle, colonne, chiavi primarie e indici. Verifica che le tabelle rilevate possano essere supportate in un'API GraphQL. Nota che per supportare la creazione di nuove righe, le tabelle necessitano di una chiave primaria, che può utilizzare più colonne. AWS AppSync mappa le colonne della tabella per digitare i campi come segue:

Tipo di dati

Tipo di campo

VARCHAR	Stringa
CHAR	Stringa
BINARY	Stringa
VARBINARY	Stringa
TINYBLOB	Stringa
TINYTEXT	Stringa
TEXT	Stringa
BLOB	Stringa
MEDIUMTEXT	Stringa
MEDIUMBLOB	Stringa
LONGTEXT	Stringa
LOBLOB	Stringa
BOOL	Booleano
BOOLEAN	Booleano
BIT	Int
TINYINT	Int
SMALLINT	Int
MEDIUMINT	Int
INT	Int
INTEGER	Int
BIGINT	Int
ANNO	Int

FLOAT	Float
DOUBLE	Float
DECIMAL	Float
DEC	Float
NUMERIC	Float
DATE	AWSDate
TIMESTAMP	Stringa
DATETIME	Stringa
TIME	AWSTime
JSON	AWSJson
ENUM	ENUM

- Una volta completata l'individuazione delle tabelle, la sezione Database verrà popolata con le informazioni dell'utente. Nella nuova sezione Tabelle del database, i dati della tabella potrebbero già essere compilati e convertiti in un tipo adatto allo schema. Se non vedi alcuni dei dati richiesti, puoi verificarli scegliendo Aggiungi tabelle, facendo clic sulle caselle di controllo relative a tali tipi nella finestra modale visualizzata, quindi scegliendo Aggiungi.

Per rimuovere un tipo dalla sezione Tabelle del database, fai clic sulla casella di controllo accanto al tipo che desideri rimuovere, quindi scegli Rimuovi. I tipi rimossi verranno inseriti nella modalità Aggiungi tabelle se desideri aggiungerli nuovamente in un secondo momento.

Nota che AWS AppSync utilizza i nomi delle tabelle come nomi di tipo, ma puoi rinominarli, ad esempio cambiando il nome di una tabella plurale simile *movies* al nome del tipo. *Movie* Per rinominare un tipo nella sezione Tabelle del database, fai clic sulla casella di controllo del tipo che desideri rinominare, quindi fai clic sull'icona a forma di matita nella colonna Nome tipo.

Per visualizzare in anteprima il contenuto dello schema in base alle tue selezioni, scegli Anteprima schema. Tieni presente che questo schema non può essere vuoto, quindi dovrai convertire almeno una tabella in un tipo. Inoltre, questo schema non può superare 1 MB di dimensione.

- In Ruolo di servizio, scegli se creare un nuovo ruolo di servizio specifico per questa importazione o utilizzare un ruolo esistente.
6. Scegli Next (Successivo).
 7. Quindi, scegli se creare un'API di sola lettura (solo query) o un'API per leggere e scrivere dati (con query e mutazioni). Quest'ultima supporta anche sottoscrizioni in tempo reale innescate da mutazioni.
 8. Scegli Next (Successivo).
 9. Controlla le tue scelte e poi scegli Crea API. AWS AppSync creerà l'API e collegherà i resolver a query e mutazioni. L'API generata è completamente operativa e può essere estesa secondo necessità.

Utilizzo della funzione di introspezione (API)

Puoi utilizzare l'API di `StartDataSourceIntrospection` introspezione per scoprire i modelli nel tuo database a livello di codice. Per maggiori dettagli sul comando, consulta [Utilizzo dell'API](#).

[StartDataSourceIntrospection](#)

Per utilizzarlo `StartDataSourceIntrospection`, fornisci il nome Amazon Resource Name (ARN) del cluster Aurora, il nome del database e l'ARN segreto. Gestione dei segreti AWS Il comando avvia il processo di introspezione. È possibile recuperare i risultati con il comando. `GetDataSourceIntrospection` È possibile specificare se il comando deve restituire la stringa Storage Definition Language (SDL) per i modelli rilevati. Ciò è utile per generare una definizione dello schema SDL direttamente dai modelli scoperti.

Ad esempio, se avete la seguente istruzione DDL (Data Definition Language) per una tabella semplice: Todos

```
create table if not exists public.todos
(
  id serial constraint todos_pk primary key,
  description text,
  due timestamp,
  "createdAt" timestamp default now()
);
```

Iniziate l'introspezione con quanto segue.


```
aws appsync start-data-source-introspection \  
  --rds-data-api-config resourceArn=<cluster-arn>,secretArn=<secret-  
arn>,databaseName=database
```

Quindi, utilizzate il `GetDataSourceIntrospection` comando per recuperare il risultato.

```
aws appsync get-data-source-introspection \  
  --introspection-id a1234567-8910-abcd-efgh-identifier \  
  --include-models-sdl
```

Ciò restituisce il seguente risultato.

```
{  
  "introspectionId": "a1234567-8910-abcd-efgh-identifier",  
  "introspectionStatus": "SUCCESS",  
  "introspectionStatusDetail": null,  
  "introspectionResult": {  
    "models": [  
      {  
        "name": "todos",  
        "fields": [  
          {  
            "name": "description",  
            "type": {  
              "kind": "Scalar",  
              "name": "String",  
              "type": null,  
              "values": null  
            },  
            "length": 0  
          },  
          {  
            "name": "due",  
            "type": {  
              "kind": "Scalar",  
              "name": "AWSDateTime",  
              "type": null,  
              "values": null  
            },  
            "length": 0  
          }  
        ],  
      }  
    ]  
  }  
}
```

```

        "name": "id",
        "type": {
          "kind": "NonNull",
          "name": null,
          "type": {
            "kind": "Scalar",
            "name": "Int",
            "type": null,
            "values": null
          },
          "values": null
        },
        "length": 0
      },
      {
        "name": "createdAt",
        "type": {
          "kind": "Scalar",
          "name": "AWSDateTime",
          "type": null,
          "values": null
        },
        "length": 0
      }
    ],
    "primaryKey": {
      "name": "PRIMARY_KEY",
      "fields": [
        "id"
      ]
    },
    "indexes": [],
    "sdl": "type todos {\n  description: String\n  due: AWSDateTime\n  nid: Int!\n  createdAt: AW\nSDatetime\n}\n"
  },
  "nextToken": null
}
}

```

Creazione di un'applicazione client utilizzando il client Amplify

Puoi connetterti alla tua API AWS AppSync GraphQL utilizzando qualsiasi client GraphQL, ma consigliamo vivamente il client Amplify v6. Amplify non solo genera automaticamente client fortemente tipizzati per l'API GraphQL, ma offre anche supporto SDKs per dati in tempo reale e funzionalità avanzate di interrogazione GraphQL nelle applicazioni client. Per le applicazioni web, Amplify può produrre un client. JavaScript Per coloro che si rivolgono ad ambienti multiplatforma o mobili, Amplify si rivolge ad Android, iOS e React Native. [Per approfondire la generazione di codice client per queste piattaforme, consulta la documentazione di Amplify.](#) Ecco una guida per iniziare il tuo viaggio con un'applicazione React: JavaScript

Note

Devi installare e configurare sia [npm](#) che [Amazon CLI](#) prima di iniziare. [Se utilizzi il client Amplify v6, segui questa guida.](#)

Per iniziare

1. Sul computer locale, accedi alla directory del progetto. Installa la libreria Amplify usando il comando seguente:

```
npm install aws-amplify
```

2. Scaricate il file di configurazione e inseritelo nella cartella del progetto. Il file di configurazione contiene in genere una `config` variabile con alcune impostazioni (endpoint, regione, modalità di autorizzazione, ecc.) definite. Ad esempio, potrebbe assomigliare a questo:

```
const config = {
  API: {
    GraphQL: {
      endpoint: 'https://abcdefghijklmnopqrstuvwxyz.appsync-api.us-west-2.amazonaws.com/graphql',
      region: 'us-west-2',
      defaultAuthMode: 'apiKey',
      apiKey: ''
    }
  }
}
```

```
    }  
  };  
  
  export default config;
```

3. Nel tuo codice, importa la libreria Amplify e la tua configurazione per configurare Amplify:

```
import { Amplify } from 'aws-amplify';  
import config from './aws-exports.js';  
  
Amplify.configure(config);
```

In alternativa, usa lo snippet nella configurazione dell'API per configurare direttamente Amplify:

```
import { Amplify } from 'aws-amplify';  
  
Amplify.configure({  
  API: {  
    GraphQL: {  
      endpoint: 'https://abcdefghijklmnopqrstuvwxyz.appsync-api.us-west-2.amazonaws.com/graphql',  
      region: 'us-west-2',  
      defaultAuthMode: 'apiKey',  
      apiKey: ''  
    }  
  }  
});
```

4. Utilizzando la toolchain Amplify, hai la possibilità di generare automaticamente le operazioni in base al tuo schema, il che ti consente di risparmiare lo sforzo di scrivere script manuali. Nella directory principale dell'applicazione, utilizzate il seguente comando CLI:

```
npx @aws-amplify/cli codegen add --apiId <id goes here> --region <region goes here>
```

Questo scaricherà lo schema dell'API e, per impostazione predefinita, genererà il codice di supporto del client nella `src/graphql` cartella. Dopo ogni implementazione dell'API, puoi eseguire nuovamente il comando seguente per generare istruzioni e tipi GraphQL aggiornati:

```
npx @aws-amplify/cli codegen
```

5. Ora puoi generare modelli per Android, Swift, Flutter e JavaScript DataStore Usa il seguente comando per scaricare lo schema:

```
aws appsync get-introspection-schema --api-id <id goes here> --region <region goes here> --format SDL schema.graphql
```

Quindi, esegui il comando seguente dalla directory principale dell'applicazione:

```
npx @aws-amplify/cli codegen models \  
--model-schema schema.graphql \  
--target [android|ios|flutter|javascript|typescript] \  
--output-dir ./
```

JavaScript tutorial resolver per AWS AppSync

Le sorgenti di dati e i resolver vengono utilizzati per AWS AppSync tradurre le richieste GraphQL e recuperare informazioni dalle tue risorse. AWS AppSync supporta il provisioning automatico e le connessioni con determinati tipi di fonti di dati. AWS AppSync supporta anche Amazon DynamoDB, AWS Lambda, database relazionali (Amazon Aurora Serverless), OpenSearch Amazon Service ed endpoint HTTP come fonti di dati. Puoi utilizzare un'API GraphQL con le tue AWS risorse esistenti o creare sorgenti di dati e resolver partendo da zero. Le sezioni seguenti hanno lo scopo di chiarire alcuni dei casi d'uso più comuni di GraphQL sotto forma di tutorial.

Argomenti

- [Creazione di una semplice applicazione post utilizzando i resolver DynamoDB JavaScript](#)
- [Utilizzo dei AWS Lambda resolver in AWS AppSync](#)
- [Utilizzo di resolver locali in AWS AppSync](#)
- [Combinazione di resolver GraphQL in AWS AppSync](#)
- [Utilizzo dei OpenSearch resolver Amazon Service in AWS AppSync](#)
- [Esecuzione di transazioni DynamoDB in AWS AppSync](#)
- [Utilizzo delle operazioni batch di DynamoDB in AWS AppSync](#)
- [Utilizzo di resolver HTTP in AWS AppSync](#)
- [Utilizzo di Aurora PostgreSQL con Data API in AWS AppSync](#)

Creazione di una semplice applicazione post utilizzando i resolver DynamoDB JavaScript

In questo tutorial, importerai le tue tabelle Amazon DynamoDB e le AWS AppSync collegherai per creare un'API GraphQL completamente funzionale JavaScript utilizzando resolver di pipeline che puoi sfruttare nella tua applicazione.

Utilizzerai la AWS AppSync console per effettuare il provisioning delle tue risorse Amazon DynamoDB, creare i tuoi resolver e collegarli alle tue fonti di dati. Potrai anche leggere e scrivere sul tuo database Amazon DynamoDB tramite istruzioni GraphQL e sottoscrivere dati in tempo reale.

Esistono passaggi specifici che devono essere completati per tradurre le istruzioni GraphQL nelle operazioni di Amazon DynamoDB e per ritradurre le risposte in GraphQL. Questo tutorial descrive il processo di configurazione attraverso diversi scenari e modelli di accesso ai dati reali.

Creazione dell'API GraphQL

Per creare un'API GraphQL in AWS AppSync

1. Apri la AppSync console e scegli Crea API.
2. Seleziona Design da zero e scegli Avanti.
3. Assegna un nome alla tua API `PostTutorialAPI`, quindi scegli Avanti. Passa alla pagina di revisione mantenendo le altre opzioni impostate sui valori predefiniti e scegli `Create`.

La AWS AppSync console crea una nuova API GraphQL per te. Per impostazione predefinita, utilizza la modalità di autenticazione tramite chiave API. Puoi usare la console per configurare ulteriormente l'API GraphQL ed eseguire query sull'API per le parti restanti di questo tutorial.

Definizione di un'API post di base

Ora che hai la tua API GraphQL, puoi configurare uno schema di base che consenta la creazione, il recupero e l'eliminazione di base dei dati post.

Per aggiungere dati allo schema

1. Nella tua API, scegli la scheda Schema.
2. Creeremo uno schema che definisce un `Post` tipo e un'operazione `addPost` per aggiungere e ottenere `Post` oggetti. Nel riquadro Schema, sostituisci il contenuto con il seguente codice:

```
schema {  
  query: Query  
  mutation: Mutation  
}  
  
type Query {  
  getPost(id: ID!): Post  
}  
  
type Mutation {  
  addPost(  
    title: String!  
    content: String!  
  ): Post  
}
```

```
        id: ID!
        author: String!
        title: String!
        content: String!
        url: String!
      ): Post!
    }

    type Post {
      id: ID!
      author: String
      title: String
      content: String
      url: String
      ups: Int!
      downs: Int!
      version: Int!
    }
```

3. Scegli Save Scheme (Salva schema).

Configurazione della tabella Amazon DynamoDB

La AWS AppSync console può aiutarti a fornire le AWS risorse necessarie per archiviare le tue risorse in una tabella Amazon DynamoDB. In questo passaggio, creerai una tabella Amazon DynamoDB per archiviare i tuoi post. Inoltre, configurerai un [indice secondario](#) che utilizzeremo in seguito.

Per creare la tua tabella Amazon DynamoDB

1. Nella pagina Schema, scegli Crea risorse.
2. Scegli Usa il tipo esistente, quindi scegli il Post tipo.
3. Nella sezione Indici aggiuntivi, scegli Aggiungi indice.
4. Assegna un nome all'indice. `author-index`
5. Imposta Primary key a `author` e la Sort chiave a `None`.
6. Disabilita la generazione automatica di GraphQL. In questo esempio, creeremo noi stessi il resolver.
7. Scegli Create (Crea).

Ora hai una nuova fonte di dati chiamata `PostTable`, che puoi vedere visitando Fonti di dati nella scheda laterale. Utilizzerai questa fonte di dati per collegare le tue query e mutazioni alla tua tabella Amazon DynamoDB.

Configurazione di un resolver `AddPost` (Amazon DynamoDB) `PutItem`

Ora che conosci AWS AppSync la tabella Amazon DynamoDB, puoi collegarla a singole query e mutazioni definendo i resolver. Il primo resolver che crei è il resolver di `addPost` pipeline utilizzato JavaScript, che ti consente di creare un post nella tua tabella Amazon DynamoDB. Un risolutore di pipeline ha i seguenti componenti:

- Posizione nello schema GraphQL per collegare il resolver. In questo caso, stai configurando un resolver nel campo `createPost` nel tipo `Mutation`. Questo resolver verrà richiamato quando il chiamante chiama `mutation`. `{ addPost(...){...} }`
- Origine dati da usare per il resolver. In questo caso, si desidera utilizzare l'origine dati DynamoDB definita in precedenza, in modo da poter aggiungere voci nella tabella DynamoDB. `post-table-for-tutorial`
- Il gestore delle richieste. Il gestore delle richieste è una funzione che gestisce la richiesta in entrata dal chiamante e la traduce in istruzioni da AWS AppSync eseguire su DynamoDB.
- Il gestore delle risposte. Il compito del gestore delle risposte è gestire la risposta da DynamoDB e tradurla nuovamente in qualcosa che GraphQL si aspetta. Questo è utile se la forma dei dati in DynamoDB è diversa rispetto al tipo `Post` in GraphQL, ma in questo caso hanno la stessa forma e di conseguenza puoi semplicemente passare i dati.

Per configurare il resolver

1. Nella tua API, scegli la scheda `Schema`.
2. Nel riquadro `Resolver`, trova il **`addPost`** campo sotto il **`Mutation`** tipo, quindi scegli `Allega`.
3. Scegli la tua fonte di dati, quindi scegli `Crea`.
4. Nel tuo editor di codice, sostituisci il codice con questo frammento:

```
import { util } from '@aws-appsync/utils'
import * as ddb from '@aws-appsync/utils/dynamodb'

export function request(ctx) {
  const item = { ...ctx.arguments, ups: 1, downs: 0, version: 1 }
  const key = { id: ctx.args.id ?? util.autoId() }
```

```
return ddb.put({ key, item })
}

export function response(ctx) {
  return ctx.result
}
```

5. Scegli Save (Salva).

Note

In questo codice, si utilizzano gli utils del modulo DynamoDB che consentono di creare facilmente richieste DynamoDB.

AWS AppSync viene fornito con un'utilità per la generazione automatica di ID chiamata `util.autoId()`, che viene utilizzata per generare un ID per il nuovo post. Se non specifichi un ID, l'utilità lo genererà automaticamente per te.

```
const key = { id: ctx.args.id ?? util.autoId() }
```

Per ulteriori informazioni sulle utilità disponibili per JavaScript, consulta [Funzionalità JavaScript di runtime per resolver e funzioni](#).

Chiama l'API per aggiungere un post

Ora che il resolver è stato configurato, AWS AppSync puoi tradurre una `addPost` mutazione in entrata in un'operazione Amazon DynamoDB. `PutItem` Puoi ora eseguire una mutazione per inserire contenuto nella tabella.

Per eseguire l'operazione

1. Nella tua API, scegli la scheda Query.
2. Nel riquadro Query, aggiungi la seguente mutazione:

```
mutation addPost {
  addPost(
    id: 123,
    author: "AUTHORNAME"
    title: "Our first post!"
  )
}
```

```

    content: "This is our first post."
    url: "https://aws.amazon.com/appsync/"
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}

```

3. Scegli Esegui (il pulsante arancione di riproduzione), quindi scegli. `addPost` I risultati del post appena creato dovrebbero apparire nel riquadro Risultati a destra del riquadro Query. La schermata visualizzata dovrebbe risultare simile a quella nell'immagine seguente:

```

{
  "data": {
    "addPost": {
      "id": "123",
      "author": "AUTHORNAME",
      "title": "Our first post!",
      "content": "This is our first post.",
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 1
    }
  }
}

```

La seguente spiegazione mostra cosa è successo:

1. AWS AppSync ha ricevuto una richiesta di `addPost` mutazione.
2. AWS AppSync esegue il gestore delle richieste del resolver. La `ddb.put` funzione crea una `PutItem` richiesta simile alla seguente:

```

{
  operation: 'PutItem',

```

```
key: { id: { S: '123' } },
attributeValues: {
  downs: { N: 0 },
  author: { S: 'AUTHORNAME' },
  ups: { N: 1 },
  title: { S: 'Our first post!' },
  version: { N: 1 },
  content: { S: 'This is our first post.' },
  url: { S: 'https://aws.amazon.com/appsync/' }
}
```

3. AWS AppSync utilizza questo valore per generare ed eseguire una richiesta Amazon PutItem DynamoDB.
4. AWS AppSync ha preso i risultati della PutItem richiesta e li ha riconvertiti in tipi GraphQL.

```
{
  "id" : "123",
  "author": "AUTHORNAME",
  "title": "Our first post!",
  "content": "This is our first post.",
  "url": "https://aws.amazon.com/appsync/",
  "ups" : 1,
  "downs" : 0,
  "version" : 1
}
```

5. Il gestore della risposta restituisce immediatamente il risultato `()return ctx.result`.
6. Il risultato finale è visibile nella risposta GraphQL.

Configurazione del resolver GetPost (Amazon DynamoDB) GetItem

Ora che sei in grado di aggiungere dati alla tabella Amazon DynamoDB, devi configurare `getPost` la query in modo che possa recuperare i dati dalla tabella. A questo scopo, devi configurare un altro resolver.

Per aggiungere il tuo resolver

1. Nella tua API, scegli la scheda Schema.
2. Nel riquadro Resolver a destra, trova il **getPost** campo relativo al **Query** tipo, quindi scegli Allega.

3. Scegli la tua fonte di dati, quindi scegli Crea.
4. Nell'editor di codice, sostituisci il codice con questo frammento:

```
import * as ddb from '@aws-appsync/utils/dynamodb'

export function request(ctx) {
  return ddb.get({ key: { id: ctx.args.id } })
}

export const response = (ctx) => ctx.result
```

5. Salva il resolver.

Note

In questo resolver, utilizziamo un'espressione della funzione freccia per il gestore delle risposte.

Chiama l'API per ricevere un post

Ora che il resolver è stato configurato, AWS AppSync sa come tradurre una `getPost` query in entrata in un'operazione Amazon DynamoDB. `GetItem` Puoi ora eseguire una query per recuperare il post creato prima.

Per eseguire la tua query

1. Nella tua API, scegli la scheda Query.
2. Nel riquadro Query, aggiungi il codice seguente e usa l'id che hai copiato dopo aver creato il post:

```
query getPost {
  getPost(id: "123") {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

```
}
}
```

3. Scegli Esegui (il pulsante arancione di riproduzione), quindi scegli. `getPost` I risultati del post appena creato dovrebbero apparire nel riquadro Risultati a destra del riquadro Query.
4. Il post recuperato da Amazon DynamoDB dovrebbe apparire nel riquadro Risultati a destra del riquadro Query. La schermata visualizzata dovrebbe risultare simile a quella nell'immagine seguente:

```
{
  "data": {
    "getPost": {
      "id": "123",
      "author": "AUTHORNAME",
      "title": "Our first post!",
      "content": "This is our first post.",
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 1
    }
  }
}
```

In alternativa, prendi il seguente esempio:

```
query getPost {
  getPost(id: "123") {
    id
    author
    title
  }
}
```

Se la tua `getPost` query richiede solo `id`, e `author` `title`, puoi modificare la funzione di richiesta per utilizzare le espressioni di proiezione per specificare solo gli attributi che desideri dalla tua tabella DynamoDB per evitare trasferimenti di dati non necessari da DynamoDB a. AWS AppSync Ad esempio, la funzione di richiesta può essere simile allo snippet riportato di seguito:

```
import * as ddb from '@aws-appsync/utils/dynamodb'
```

```
export function request(ctx) {
  return ddb.get({
    key: { id: ctx.args.id },
    projection: ['author', 'id', 'title'],
  })
}

export const response = (ctx) => ctx.result
```

Puoi anche usare un [selectionSetList](#) with `getPost` per rappresentare: expression

```
import * as ddb from '@aws-appsync/utils/dynamodb'

export function request(ctx) {
  const projection = ctx.info.selectionSetList.map((field) => field.replace('/', '.'))
  return ddb.get({ key: { id: ctx.args.id }, projection })
}

export const response = (ctx) => ctx.result
```

Creare una mutazione UpdatePost (Amazon DynamoDB) UpdateItem

Finora, puoi creare e recuperare Post oggetti in Amazon DynamoDB. Successivamente, imposterai una nuova mutazione per aggiornare un oggetto. Rispetto alla `addPost` mutazione che richiede la specificazione di tutti i campi, questa mutazione consente di specificare solo i campi che si desidera modificare. Ha inoltre introdotto un nuovo `expectedVersion` argomento che consente di specificare la versione che si desidera modificare. Imposterete una condizione che assicurerà che stiate modificando la versione più recente dell'oggetto. Potrai farlo utilizzando `UpdateItem` Amazon DynamoDB operation.sc

Per aggiornare il resolver

1. Nella tua API, scegli la scheda Schema.
2. Nel riquadro Schema, modificare il tipo `Mutation` per aggiungere una nuova mutazione `updatePost` come segue:

```
type Mutation {
  updatePost(
    id: ID!,
    author: String,
```

```

        title: String,
        content: String,
        url: String,
        expectedVersion: Int!
    ): Post

    addPost(
        id: ID!
        author: String!
        title: String!
        content: String!
        url: String!
    ): Post!
}
```

3. Scegli Save Scheme (Salva schema).

4. Nel riquadro Resolver a destra, trova il **updatePost** campo appena creato relativo al **Mutation** tipo, quindi scegli Allega. Crea il tuo nuovo resolver usando lo snippet qui sotto:

```

import { util } from '@aws-appsync/utils';
import * as ddb from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  const { id, expectedVersion, ...rest } = ctx.args;
  const values = Object.entries(rest).reduce((obj, [key, value]) => {
    obj[key] = value ?? ddb.operations.remove();
    return obj;
  }, {});

  return ddb.update({
    key: { id },
    condition: { version: { eq: expectedVersion } },
    update: { ...values, version: ddb.operations.increment(1) },
  });
}

export function response(ctx) {
  const { error, result } = ctx;
  if (error) {
    util.appendError(error.message, error.type);
  }
  return result;
}
```


5. Salva tutte le modifiche apportate.

Questo resolver viene utilizzato `ddb.update` per creare una richiesta Amazon DynamoDB. `UpdateItem` Invece di scrivere l'intero articolo, stai semplicemente chiedendo ad Amazon DynamoDB di aggiornare determinati attributi. Questa operazione viene eseguita utilizzando le espressioni di aggiornamento di Amazon DynamoDB.

La `ddb.update` funzione accetta una chiave e un oggetto di aggiornamento come argomenti. Quindi, si controllano i valori degli argomenti in entrata. Quando un valore è impostato su `null`, utilizzate l'operazione `remove` DynamoDB per segnalare che il valore deve essere rimosso dall'elemento DynamoDB.

C'è anche una nuova sezione. `condition` Un'espressione di condizione consente di indicare AWS AppSync ad Amazon DynamoDB se la richiesta deve avere successo o meno in base allo stato dell'oggetto già in Amazon DynamoDB prima dell'esecuzione dell'operazione. In questo caso, vuoi che la `UpdateItem` richiesta abbia esito positivo solo se il `version` campo dell'elemento attualmente in Amazon DynamoDB corrisponde `expectedVersion` esattamente all'argomento. Quando l'elemento viene aggiornato, vogliamo incrementare il valore di `version`. Questo è facile da fare con la funzione `increment` operativa.

Per ulteriori informazioni sulle espressioni condizionali, consulta la documentazione sulle [espressioni di condizione](#).

Per maggiori informazioni sulla `UpdateItem` richiesta, consulta la [UpdateItem](#) documentazione e la documentazione del modulo [DynamoDB](#).

Per ulteriori informazioni su come scrivere espressioni di aggiornamento, consulta la documentazione di [UpdateExpressionsDynamoDB](#).

Chiama l'API per aggiornare un post

Proviamo ad aggiornare l'Post oggetto con il nuovo resolver.

Per aggiornare l'oggetto

1. Nella tua API, scegli la scheda Query.
2. Nel riquadro Query, aggiungi la seguente mutazione. Dovrai anche aggiornare l'id argomento al valore annotato in precedenza:

```
mutation updatePost {
```

```
updatePost(  
  id:123  
  title: "An empty story"  
  content: null  
  expectedVersion: 1  
) {  
  id  
  author  
  title  
  content  
  url  
  ups  
  downs  
  version  
}  
}
```

3. Scegli Esegui (il pulsante arancione di riproduzione), quindi scegli `updatePost`.
4. Il post aggiornato in Amazon DynamoDB dovrebbe apparire nel riquadro Risultati a destra del riquadro Query. La schermata visualizzata dovrebbe risultare simile a quella nell'immagine seguente:

```
{  
  "data": {  
    "updatePost": {  
      "id": "123",  
      "author": "A new author",  
      "title": "An empty story",  
      "content": null,  
      "url": "https://aws.amazon.com/appsync/",  
      "ups": 1,  
      "downs": 0,  
      "version": 2  
    }  
  }  
}
```

In questa richiesta, hai chiesto ad Amazon DynamoDB di aggiornare `title` solo i campi AWS AppSync and. `content` Tutti gli altri campi sono stati lasciati soli (a parte l'incremento del `version` campo). Hai impostato l'`title` attributo su un nuovo valore e rimosso l'`content` attributo dal post. I campi `author`, `url`, `ups` e `downs` sono stati lasciati invariati. Prova a eseguire nuovamente la

richiesta di mutazione lasciando la richiesta esattamente com'è. Noterai una risposta simile alla seguente:

```
{
  "data": {
    "updatePost": null
  },
  "errors": [
    {
      "path": [
        "updatePost"
      ],
      "data": null,
      "errorType": "DynamoDB:ConditionalCheckFailedException",
      "errorInfo": null,
      "locations": [
        {
          "line": 2,
          "column": 3,
          "sourceName": null
        }
      ],
      "message": "The conditional request failed (Service: DynamoDb, Status Code: 400, Request ID: 1RR3QN5F35CS8IV5VR40Q09NNBVV4KQNS05AEMVJF66Q9ASUAAJG)"
    }
  ]
}
```

La richiesta fallisce perché l'espressione della condizione restituisce: `false`

1. La prima volta che hai eseguito la richiesta, il valore del `version` campo del post in Amazon DynamoDB 1 era corrispondente all'argomento. `expectedVersion` La richiesta è riuscita, il che significa che il `version` campo è stato incrementato in Amazon DynamoDB a. 2
2. La seconda volta che hai eseguito la richiesta, il valore del `version` campo del post in Amazon DynamoDB 2 era, che non corrispondeva all'argomento. `expectedVersion`

Questo modello viene in genere chiamato blocco ottimistico.

Crea mutazioni di voto (Amazon DynamoDB UpdateItem)

Il `Post` tipo contiene `downs` campi per consentire la registrazione di voti positivi `ups` e negativi. Tuttavia, al momento, l'API non ci consente di fare nulla con loro. Aggiungiamo una mutazione per consentirci di votare positivamente e negativamente i post.

Per aggiungere la tua mutazione

1. Nella tua API, scegli la scheda Schema.
2. Nel riquadro Schema, modifica il `Mutation` tipo e aggiungi l'`DIRECTIONenum` per aggiungere nuove mutazioni di voto:

```
type Mutation {  
  vote(id: ID!, direction: DIRECTION!): Post  
  updatePost(  
    id: ID!,  
    author: String,  
    title: String,  
    content: String,  
    url: String,  
    expectedVersion: Int!  
  ): Post  
  addPost(  
    id: ID,  
    author: String!,  
    title: String!,  
    content: String!,  
    url: String!  
  ): Post!  
}  
  
enum DIRECTION {  
  UP  
  DOWN  
}
```

3. Scegli `Save Scheme` (Salva schema).
4. Nel riquadro Resolver a destra, individuate il **`vote`** campo appena creato relativo al **`Mutation`** tipo, quindi scegliete `Allega`. Crea un nuovo resolver creando e sostituendo il codice con il seguente frammento:

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  const field = ctx.args.direction === 'UP' ? 'ups' : 'downs';
  return ddb.update({
    key: { id: ctx.args.id },
    update: {
      [field]: ddb.operations.increment(1),
      version: ddb.operations.increment(1),
    },
  });
}

export const response = (ctx) => ctx.result;
```

5. Salva tutte le modifiche apportate.

Chiama l'API per dare un voto positivo o negativo a un post

Ora che i nuovi resolver sono stati configurati, AWS AppSync sa come tradurre una modifica `upvotePost` o una `downvote` mutazione in entrata in un'operazione Amazon DynamoDB. `UpdateItem` Ora puoi eseguire mutazioni per assegnare voti positivi e voti negativi al post creato prima.

Per eseguire la tua mutazione

1. Nella tua API, scegli la scheda Query.
2. Nel riquadro Query, aggiungi la seguente mutazione. Dovrai anche aggiornare l'idargomento al valore annotato in precedenza:

```
mutation votePost {
  vote(id:123, direction: UP) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

```
}
}
```

3. Scegli Esegui (il pulsante arancione di riproduzione), quindi scegli `votePost`.
4. Il post aggiornato in Amazon DynamoDB dovrebbe apparire nel riquadro Risultati a destra del riquadro Query. La schermata visualizzata dovrebbe risultare simile a quella nell'immagine seguente:

```
{
  "data": {
    "vote": {
      "id": "123",
      "author": "A new author",
      "title": "An empty story",
      "content": null,
      "url": "https://aws.amazon.com/appsync/",
      "ups": 6,
      "downs": 0,
      "version": 4
    }
  }
}
```

5. Scegli Esegui ancora un paio di volte. Dovresti vedere i `version` campi `ups` and incrementare 1 ogni volta che esegui la query.
6. Cambia la query per chiamarla con un'altra `DIRECTION`.

```
mutation votePost {
  vote(id:123, direction: DOWN) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

7. Scegli Esegui (il pulsante di riproduzione arancione), quindi scegli `votePost`.

Questa volta, dovresti vedere i `version` campi `downs` and incrementarsi 1 ogni volta che esegui la query.

Configurazione di un resolver `DeletePost` (Amazon DynamoDB) `DeleteItem`

Successivamente, ti consigliamo di creare una mutazione per eliminare un post. A tale scopo, utilizzerai l'operazione `DeleteItem` Amazon DynamoDB.

Per aggiungere la tua mutazione

1. Nel tuo schema, scegli la scheda `Schema`.
2. Nel riquadro `Schema`, modifica il `Mutation` tipo per aggiungere una nuova `deletePost` mutazione:

```
type Mutation {  
  deletePost(id: ID!, expectedVersion: Int): Post  
  vote(id: ID!, direction: DIRECTION!): Post  
  updatePost(  
    id: ID!,  
    author: String!,  
    title: String!,  
    content: String!,  
    url: String!,  
    expectedVersion: Int!  
  ): Post  
  addPost(  
    id: ID!  
    author: String!,  
    title: String!,  
    content: String!,  
    url: String!  
  ): Post!  
}
```

3. Questa volta, hai reso il `expectedVersion` campo facoltativo. Quindi, scegli `Salva schema`.
4. Nel riquadro `Resolver` a destra, trova il **`delete`** campo appena creato nel **`Mutation`** tipo, quindi scegli `Allega`. Crea un nuovo resolver utilizzando il seguente codice:

```
import { util } from '@aws-appsync/utils'
```

```
import { util } from '@aws-appsync/utils';
import * as ddb from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  let condition = null;
  if (ctx.args.expectedVersion) {
    condition = {
      or: [
        { id: { attributeExists: false } },
        { version: { eq: ctx.args.expectedVersion } },
      ],
    };
  }
  return ddb.remove({ key: { id: ctx.args.id }, condition });
}

export function response(ctx) {
  const { error, result } = ctx;
  if (error) {
    util.appendError(error.message, error.type);
  }
  return result;
}
```

Note

L'`expectedVersion` argomento è un argomento facoltativo. Se il chiamante imposta un `expectedVersion` argomento nella richiesta, il gestore della richiesta aggiunge una condizione che consente alla `DeleteItem` richiesta di avere successo solo se l'elemento è già stato eliminato o se l'`version` attributo del post in Amazon DynamoDB corrisponde esattamente a `expectedVersion`. Se non viene inserito, nessuna espressione di condizione viene specificata nella richiesta `DeleteItem`. Ha successo indipendentemente dal valore o dall'esistenza `version` o meno dell'elemento in Amazon DynamoDB. Anche se stai eliminando un articolo, puoi restituire l'elemento che è stato eliminato, se non lo era già.

Per maggiori informazioni sulla `DeleteItem` richiesta, consulta la [DeleteItem](#) documentazione.

Chiama l'API per eliminare un post

Ora che il resolver è stato configurato, AWS AppSync sa come tradurre una delete mutazione in entrata in un'operazione Amazon DynamoDB. DeleteItem Puoi ora eseguire una mutazione per eliminare qualcosa nella tabella.

Per eseguire la tua mutazione

1. Nella tua API, scegli la scheda Query.
2. Nel riquadro Query, aggiungi la seguente mutazione. Dovrai anche aggiornare l'idargomento al valore annotato in precedenza:

```
mutation deletePost {  
  deletePost(id:123) {  
    id  
    author  
    title  
    content  
    url  
    ups  
    downs  
    version  
  }  
}
```

3. Scegli Esegui (il pulsante arancione di riproduzione), quindi scegli deletePost.
4. Il post viene eliminato da Amazon DynamoDB. Tieni presente che AWS AppSync restituisce il valore dell'elemento che è stato eliminato da Amazon DynamoDB, che dovrebbe apparire nel riquadro Risultati a destra del riquadro Query. La schermata visualizzata dovrebbe risultare simile a quella nell'immagine seguente:

```
{  
  "data": {  
    "deletePost": {  
      "id": "123",  
      "author": "A new author",  
      "title": "An empty story",  
      "content": null,  
      "url": "https://aws.amazon.com/appsync/",  
      "ups": 6,  
      "downs": 4,  
    }  
  }  
}
```

```
    "version": 12
  }
}
```

5. Il valore viene restituito solo se questa chiamata a `deletePost` è quella che lo elimina effettivamente da Amazon DynamoDB. Scegli nuovamente Esegui.
6. La chiamata riesce ancora, ma non viene restituito alcun valore:

```
{
  "data": {
    "deletePost": null
  }
}
```

7. Ora, proviamo a eliminare un post, ma questa volta specificando un `expectedValue`. Innanzitutto, devi creare un nuovo post perché hai appena eliminato quello con cui hai lavorato finora.
8. Nel riquadro Query, aggiungi la seguente mutazione:

```
mutation addPost {
  addPost(
    id:123
    author: "AUTHORNAME"
    title: "Our second post!"
    content: "A new post."
    url: "https://aws.amazon.com/appsync/"
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

9. Scegli Esegui (il pulsante arancione di riproduzione), quindi scegli. `addPost`

10 I risultati del post appena creato dovrebbero apparire nel riquadro Risultati a destra del riquadro Query. Registra `id` l'oggetto appena creato perché ti servirà in un attimo. La schermata visualizzata dovrebbe risultare simile a quella nell'immagine seguente:

```
{
  "data": {
    "addPost": {
      "id": "123",
      "author": "AUTHORNAME",
      "title": "Our second post!",
      "content": "A new post.",
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 1
    }
  }
}
```

11 Ora, proviamo a eliminare quel post con un valore illegale per `ExpectedVersion`. Nel riquadro Query, aggiungi la seguente mutazione. Dovrai anche aggiornare l'`id` d'argomento al valore annotato in precedenza:

```
mutation deletePost {
  deletePost(
    id:123
    expectedVersion: 9999
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

12 Scegli Esegui (il pulsante arancione di riproduzione), quindi scegli `deletePost`. Viene restituito il seguente risultato:

```
{
  "data": {
    "deletePost": null
  },
  "errors": [
    {
      "path": [
        "deletePost"
      ],
      "data": null,
      "errorType": "DynamoDB:ConditionalCheckFailedException",
      "errorInfo": null,
      "locations": [
        {
          "line": 2,
          "column": 3,
          "sourceName": null
        }
      ],
      "message": "The conditional request failed (Service: DynamoDb, Status Code: 400, Request ID: 70830037M1FTFRK038A4CI9H43VV4KQNS05AEMVJF66Q9ASUAAJG)"
    }
  ]
}
```

13 La richiesta non è riuscita perché l'espressione della condizione restituisce `false`. Il valore `version` del post in Amazon DynamoDB non corrisponde a quello specificato `expectedValue` negli argomenti. Il valore corrente dell'oggetto viene restituito nel campo `data` nella sezione `errors` della risposta GraphQL. Riprova la richiesta, correggendo il valore di `expectedVersion`:

```
mutation deletePost {
  deletePost(
    id:123
    expectedVersion: 1
  ) {
    id
    author
    title
    content
    url
    ups
    downs
  }
}
```

```

    version
  }
}
```

14. Scegli Esegui (il pulsante arancione di riproduzione), quindi scegli. deletePost

Questa volta la richiesta ha esito positivo e viene restituito il valore che è stato eliminato da Amazon DynamoDB:

```

{
  "data": {
    "deletePost": {
      "id": "123",
      "author": "AUTHORNAME",
      "title": "Our second post!",
      "content": "A new post.",
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 1
    }
  }
}
```

15. Scegli di nuovo Esegui. La chiamata riesce ancora, ma questa volta non viene restituito alcun valore perché il post è già stato eliminato in Amazon DynamoDB.

```
{ "data": { "deletePost": null } }
```

Configurazione di un resolver AllPost (Amazon DynamoDB Scan)

Finora, l'API è utile solo se conosci ogni post che vuoi guardare. id Aggiungiamo ora un nuovo resolver che restituisce tutti i post nella tabella.

Per aggiungere la tua mutazione

1. Nella tua API, scegli la scheda Schema.
2. Nel riquadro Schema, modificare il tipo Query per aggiungere una nuova query allPost come segue:

```
type Query {
```

```
allPost(limit: Int, nextToken: String): PaginatedPosts!  
getPost(id: ID): Post  
}
```

3. Aggiungi un nuovo tipo `PaginationPosts`:

```
type PaginatedPosts {  
  posts: [Post!]!  
  nextToken: String  
}
```

4. Scegli `Save Scheme` (Salva schema).

5. Nel riquadro Resolver a destra, trova il **`allPost`** campo appena creato nel **`Query`** tipo, quindi scegli `Allega`. Crea un nuovo resolver con il seguente codice:

```
import * as ddb from '@aws-appsync/utils/dynamodb';  
  
export function request(ctx) {  
  const { limit = 20, nextToken } = ctx.arguments;  
  return ddb.scan({ limit, nextToken });  
}  
  
export function response(ctx) {  
  const { items: posts = [], nextToken } = ctx.result;  
  return { posts, nextToken };  
}
```

Il gestore delle richieste di questo resolver prevede due argomenti opzionali:

- `limit`- Specifica il numero massimo di elementi da restituire in una singola chiamata.
- `nextToken`- Utilizzato per recuperare il prossimo set di risultati (mostreremo da dove `nextToken` proviene il valore di in seguito).

6. Salva tutte le modifiche apportate al tuo resolver.

Per ulteriori informazioni sulla `Scan` richiesta, consulta la documentazione di riferimento di [Scan](#).

Chiama l'API per scansionare tutti i post

Ora che il resolver è stato configurato, AWS AppSync sa come tradurre una `allPost` query in entrata in un'operazione Amazon DynamoDB. `Scan` Puoi ora analizzare la tabella per recuperare tutti

i post. Prima di provare, devi immettere nella tabella alcuni dati, perché hai eliminato tutti quelli usati finora.

Per aggiungere e interrogare dati

1. Nella tua API, scegli la scheda Query.
2. Nel riquadro Query, aggiungi la seguente mutazione:

```
mutation addPost {  
  post1: addPost(id:1 author: "AUTHORNAME" title: "A series of posts, Volume 1"  
content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }  
  post2: addPost(id:2 author: "AUTHORNAME" title: "A series of posts, Volume 2"  
content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }  
  post3: addPost(id:3 author: "AUTHORNAME" title: "A series of posts, Volume 3"  
content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }  
  post4: addPost(id:4 author: "AUTHORNAME" title: "A series of posts, Volume 4"  
content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }  
  post5: addPost(id:5 author: "AUTHORNAME" title: "A series of posts, Volume 5"  
content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }  
  post6: addPost(id:6 author: "AUTHORNAME" title: "A series of posts, Volume 6"  
content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }  
  post7: addPost(id:7 author: "AUTHORNAME" title: "A series of posts, Volume 7"  
content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }  
  post8: addPost(id:8 author: "AUTHORNAME" title: "A series of posts, Volume 8"  
content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }  
  post9: addPost(id:9 author: "AUTHORNAME" title: "A series of posts, Volume 9"  
content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }  
}
```

3. Scegli Esegui (il pulsante di riproduzione arancione).
4. Analizziamo ora la tabella, restituendo cinque risultati per volta. Nel riquadro Interrogazioni, aggiungi la seguente interrogazione:

```
query allPost {  
  allPost(limit: 5) {  
    posts {  
      id  
      title  
    }  
    nextToken  
  }  
}
```

5. Scegli Esegui (il pulsante di riproduzione arancione), quindi scegli `allPost`.

I primi cinque post dovrebbero apparire nel riquadro Risultati a destra del riquadro Query. La schermata visualizzata dovrebbe risultare simile a quella nell'immagine seguente:

```
{
  "data": {
    "allPost": {
      "posts": [
        {
          "id": "5",
          "title": "A series of posts, Volume 5"
        },
        {
          "id": "1",
          "title": "A series of posts, Volume 1"
        },
        {
          "id": "6",
          "title": "A series of posts, Volume 6"
        },
        {
          "id": "9",
          "title": "A series of posts, Volume 9"
        },
        {
          "id": "7",
          "title": "A series of posts, Volume 7"
        }
      ],
      "nextToken": "<token>"
    }
  }
}
```

6. Hai ricevuto cinque risultati e uno `nextToken` che puoi utilizzare per ottenere il prossimo set di risultati. Aggiornare la query `allPost` in modo da includere `nextToken` dal set precedente di risultati:

```
query allPost {
  allPost(
    limit: 5
    nextToken: "<token>"
  )
}
```



```
) {  
  posts {  
    id  
    author  
  }  
  nextToken  
}  
}
```

7. Scegli Esegui (il pulsante arancione di riproduzione), quindi scegli `allPost`.

I quattro post rimanenti dovrebbero apparire nel riquadro Risultati a destra del riquadro Interrogazioni. Non c'è nessuno `nextToken` in questo set di risultati perché hai sfogliato tutti e nove i post senza che ne rimanga nessuno. La schermata visualizzata dovrebbe risultare simile a quella nell'immagine seguente:

```
{  
  "data": {  
    "allPost": {  
      "posts": [  
        {  
          "id": "2",  
          "title": "A series of posts, Volume 2"  
        },  
        {  
          "id": "3",  
          "title": "A series of posts, Volume 3"  
        },  
        {  
          "id": "4",  
          "title": "A series of posts, Volume 4"  
        },  
        {  
          "id": "8",  
          "title": "A series of posts, Volume 8"  
        }  
      ],  
      "nextToken": null  
    }  
  }  
}
```

Configurazione di un allPostsBy Author resolver (Amazon DynamoDB Query)

Oltre a scansionare Amazon DynamoDB per tutti i post, puoi anche interrogare Amazon DynamoDB per recuperare i post creati da un autore specifico. La tabella Amazon DynamoDB creata in precedenza contiene già `GlobalSecondaryIndex` una `author-index` chiamata che puoi usare con un'operazione Amazon DynamoDB per recuperare tutti i Query post creati da un autore specifico.

Per aggiungere la tua query

1. Nella tua API, scegli la scheda Schema.
2. Nel riquadro Schema, modificare il tipo Query per aggiungere una nuova query `allPostsByAuthor` come segue:

```
type Query {  
  allPostsByAuthor(author: String!, limit: Int, nextToken: String): PaginatedPosts!  
  allPost(limit: Int, nextToken: String): PaginatedPosts!  
  getPost(id: ID): Post  
}
```

Tieni presente che utilizza lo stesso `PaginatedPosts` tipo che hai usato con la `allPost` query.

3. Scegli `Save Scheme` (Salva schema).
4. Nel riquadro Resolver a destra, individuate il **`allPostsByAuthor`** campo appena creato relativo al **Query** tipo, quindi scegliete `Allega`. Crea un resolver utilizzando lo snippet seguente:

```
import * as ddb from '@aws-appsync/utils/dynamodb';  
  
export function request(ctx) {  
  const { limit = 20, nextToken, author } = ctx.arguments;  
  return ddb.query({  
    index: 'author-index',  
    query: { author: { eq: author } },  
    limit,  
    nextToken,  
  });  
}  
  
export function response(ctx) {
```

```
const { items: posts = [], nextToken } = ctx.result;
return { posts, nextToken };
}
```

Come il `allPost` resolver, questo resolver ha due argomenti opzionali:

- `limit`- Specifica il numero massimo di elementi da restituire in una singola chiamata.
- `nextToken`- Recupera il successivo set di risultati (il valore di `nextToken` può essere ottenuto da una chiamata precedente).

5. Salva tutte le modifiche apportate al tuo resolver.

Per ulteriori informazioni sulla Query richiesta, consulta la documentazione di riferimento di [Query](#).

Chiama l'API per interrogare tutti i post per autore

Ora che il resolver è stato configurato, AWS AppSync sa come tradurre una `allPostsByAuthor` mutazione in entrata in un'operazione DynamoDB sull'indice. Query `author-index` Puoi ora eseguire una query sulla tabella per recuperare tutti i post di un autore specifico.

Prima di questo, tuttavia, riempiamo la tabella con altri post, perché finora tutti i post hanno lo stesso autore.

Per aggiungere dati e interrogare

1. Nella tua API, scegli la scheda Query.
2. Nel riquadro Query, aggiungi la seguente mutazione:

```
mutation addPost {
  post1: addPost(id:10 author: "Nadia" title: "The cutest dog in the world" content:
    "So cute. So very, very cute." url: "https://aws.amazon.com/appsync/" ) { author,
    title }
  post2: addPost(id:11 author: "Nadia" title: "Did you know...?" content: "AppSync
    works offline?" url: "https://aws.amazon.com/appsync/" ) { author, title }
  post3: addPost(id:12 author: "Steve" title: "I like GraphQL" content: "It's great"
    url: "https://aws.amazon.com/appsync/" ) { author, title }
}
```

3. Scegli Esegui (il pulsante arancione di riproduzione), quindi scegli. `addPost`
4. Esegui ora la query sulla tabella, per restituire tutti i post il cui autore è Nadia. Nel riquadro Interrogazioni, aggiungi la seguente interrogazione:

```
query allPostsByAuthor {
  allPostsByAuthor(author: "Nadia") {
    posts {
      id
      title
    }
    nextToken
  }
}
```

5. Scegli Esegui (il pulsante di riproduzione arancione), quindi scegli `allPostsByAuthor`. Tutti i post creati da Nadia dovrebbero apparire nel riquadro Risultati a destra del riquadro Query. La schermata visualizzata dovrebbe risultare simile a quella nell'immagine seguente:

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world"
        },
        {
          "id": "11",
          "title": "Did you know...?"
        }
      ],
      "nextToken": null
    }
  }
}
```

6. La paginazione funziona per Query esattamente come per Scan. Ad esempio, cerchiamo tutti i post di `AUTHORNAME`, restituendone cinque per volta.
7. Nel riquadro Interrogazioni, aggiungi la seguente query:

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "AUTHORNAME"
    limit: 5
  ) {
```

```
    posts {
      id
      title
    }
    nextToken
  }
}
```

8. Scegli Esegui (il pulsante di riproduzione arancione), quindi scegli `allPostsByAuthor`. Tutti i post creati da `AUTHORNAME` dovrebbero apparire nel riquadro Risultati a destra del riquadro Query. La schermata visualizzata dovrebbe risultare simile a quella nell'immagine seguente:

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "6",
          "title": "A series of posts, Volume 6"
        },
        {
          "id": "4",
          "title": "A series of posts, Volume 4"
        },
        {
          "id": "2",
          "title": "A series of posts, Volume 2"
        },
        {
          "id": "7",
          "title": "A series of posts, Volume 7"
        },
        {
          "id": "1",
          "title": "A series of posts, Volume 1"
        }
      ],
      "nextToken": "<token>"
    }
  }
}
```

9. Aggiornare l'argomento `nextToken` con il valore restituito dalla query precedente, come segue:

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "AUTHORNAME"
    limit: 5
    nextToken: "<token>"
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

10. Scegli Esegui (il pulsante arancione di riproduzione), quindi scegli. `allPostsByAuthor`. I post rimanenti creati da `AUTHORNAME` dovrebbero apparire nel riquadro Risultati a destra del riquadro Query. La schermata visualizzata dovrebbe risultare simile a quella nell'immagine seguente:

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "8",
          "title": "A series of posts, Volume 8"
        },
        {
          "id": "5",
          "title": "A series of posts, Volume 5"
        },
        {
          "id": "3",
          "title": "A series of posts, Volume 3"
        },
        {
          "id": "9",
          "title": "A series of posts, Volume 9"
        }
      ],
      "nextToken": null
    }
  }
}
```

```
}
```

Utilizzo dei set

Fino a questo punto, il `Post` tipo era un key/value oggetto piatto. Puoi anche modellare oggetti complessi con il tuo resolver, come set, elenchi e mappe. Aggiorniamo ora il tipo `Post` perché includa tag. Un post può avere zero o più tag, che vengono archiviati in DynamoDB come set di stringhe. Dobbiamo anche configurare alcune mutazioni per aggiungere e rimuovere tag e una query per analizzare i post con un tag specifico.

Per configurare i tuoi dati

1. Nella tua API, scegli la scheda Schema.
2. Nel riquadro Schema, modificare il tipo `Post` per aggiungere un nuovo campo `tags` come segue:

```
type Post {  
  id: ID!  
  author: String  
  title: String  
  content: String  
  url: String  
  ups: Int!  
  downs: Int!  
  version: Int!  
  tags: [String!]  
}
```

3. Nel riquadro Schema, modificare il tipo `Query` per aggiungere una nuova query `allPostsByTag` come segue:

```
type Query {  
  allPostsByTag(tag: String!, limit: Int, nextToken: String): PaginatedPosts!  
  allPostsByAuthor(author: String!, limit: Int, nextToken: String): PaginatedPosts!  
  allPost(limit: Int, nextToken: String): PaginatedPosts!  
  getPost(id: ID): Post  
}
```

4. Nel riquadro Schema, modifica il `Mutation` tipo per aggiungere nuove `addTag` e `removeTag` mutazioni come segue:

```

type Mutation {
  addTag(id: ID!, tag: String!): Post
  removeTag(id: ID!, tag: String!): Post
  deletePost(id: ID!, expectedVersion: Int): Post
  upvotePost(id: ID!): Post
  downvotePost(id: ID!): Post
  updatePost(
    id: ID!,
    author: String,
    title: String,
    content: String,
    url: String,
    expectedVersion: Int!
  ): Post
  addPost(
    author: String!,
    title: String!,
    content: String!,
    url: String!
  ): Post!
}

```

5. Scegli Save Scheme (Salva schema).

6. Nel riquadro Resolver a destra, individuate il **allPostsByTag** campo appena creato relativo al **Query** tipo, quindi scegliete Allega. Crea il tuo resolver usando lo snippet qui sotto:

```

import * as ddb from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  const { limit = 20, nextToken, tag } = ctx.arguments;
  return ddb.scan({ limit, nextToken, filter: { tags: { contains: tag } } });
}

export function response(ctx) {
  const { items: posts = [], nextToken } = ctx.result;
  return { posts, nextToken };
}

```

7. Salva tutte le modifiche che hai apportato al tuo resolver.

8. Ora, fai lo stesso per il Mutation campo addTag usando lo snippet qui sotto:

 Note

Sebbene le utilità DynamoDB attualmente non supportino le operazioni sui set, puoi comunque interagire con i set creando tu stesso la richiesta.

```
import { util } from '@aws-appsync/utils'

export function request(ctx) {
  const { id, tag } = ctx.arguments
  const expressionValues = util.dynamodb.toMapValues({ ':plusOne': 1 })
  expressionValues[':tags'] = util.dynamodb.toStringSet([tag])

  return {
    operation: 'UpdateItem',
    key: util.dynamodb.toMapValues({ id }),
    update: {
      expression: `ADD tags :tags, version :plusOne`,
      expressionValues,
    },
  }
}

export const response = (ctx) => ctx.result
```

9. Salva tutte le modifiche apportate al tuo resolver.

10. Ripeti l'operazione ancora una volta per il `Mutation` campo `removeTag` utilizzando lo snippet seguente:

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { id, tag } = ctx.arguments;
  const expressionValues = util.dynamodb.toMapValues({ ':plusOne': 1 });
  expressionValues[':tags'] = util.dynamodb.toStringSet([tag]);

  return {
    operation: 'UpdateItem',
    key: util.dynamodb.toMapValues({ id }),
    update: {
```

```

        expression: `DELETE tags :tags ADD version :plusOne`,
        expressionValues,
      },
    };
  }

  export const response = (ctx) => ctx.result.export

```

11. Salva tutte le modifiche apportate al tuo resolver.

Chiamata dell'API per usare tag

Ora che hai configurato i resolver, AWS AppSync sa come tradurre le richieste in entrata `addTag` e le richieste `allPostsByTag` in DynamoDB e nelle operazioni. `removeTag` `UpdateItem` `Scan` Per provare, selezioniamo uno dei post creati prima. Ad esempio, è possibile utilizzare un post redatto dall'utente Nadia.

Per usare i tag

1. Nella tua API, scegli la scheda Query.
2. Nel riquadro Query, aggiungi la seguente query:

```

query allPostsByAuthor {
  allPostsByAuthor(
    author: "Nadia"
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}

```

3. Scegli Esegui (il pulsante di riproduzione arancione), quindi scegli `allPostsByAuthor`.
4. Tutti i post di Nadia dovrebbero apparire nel riquadro Risultati a destra del riquadro Query. La schermata visualizzata dovrebbe risultare simile a quella nell'immagine seguente:

```

{
  "data": {
    "allPostsByAuthor": {

```

```

    "posts": [
      {
        "id": "10",
        "title": "The cutest dog in the world"
      },
      {
        "id": "11",
        "title": "Did you known...?"
      }
    ],
    "nextToken": null
  }
}

```

5. Usiamo quello con il titolo Il cane più carino del mondo. Registralo `id` perché lo userai più tardi. Ora proviamo ad aggiungere un dog tag.
6. Nel riquadro Query, aggiungi la seguente mutazione. Dovremo anche aggiornare l'argomento `id` sul valore che abbiamo annotato in precedenza.

```

mutation addTag {
  addTag(id:10 tag: "dog") {
    id
    title
    tags
  }
}

```

7. Scegli Esegui (il pulsante arancione di riproduzione), quindi scegli. `addTag` Il post viene aggiornato con il nuovo tag:

```

{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog"
      ]
    }
  }
}

```

8. Puoi aggiungere altri tag. Aggiorna la mutazione per cambiare l'argomento in puppy:

```
mutation addTag {  
  addTag(id:10 tag: "puppy") {  
    id  
    title  
    tags  
  }  
}
```

9. Scegli Esegui (il pulsante arancione di riproduzione), quindi scegli addTag. Il post viene aggiornato con il nuovo tag:

```
{  
  "data": {  
    "addTag": {  
      "id": "10",  
      "title": "The cutest dog in the world",  
      "tags": [  
        "dog",  
        "puppy"  
      ]  
    }  
  }  
}
```

10. Puoi anche eliminare i tag. Nel riquadro Query, aggiungi la seguente mutazione. Dovrai anche aggiornare l'argomento al valore annotato in precedenza:

```
mutation removeTag {  
  removeTag(id:10 tag: "puppy") {  
    id  
    title  
    tags  
  }  
}
```

11. Scegli Esegui (il pulsante arancione di riproduzione), quindi scegli removeTag. Il post viene aggiornato e il tag puppy viene eliminato.

```
{  
  "data": {
```

```
"addTag": {  
  "id": "10",  
  "title": "The cutest dog in the world",  
  "tags": [  
    "dog"  
  ]  
}  
}  
}
```

12 Puoi anche cercare tutti i post che hanno un tag. Nel riquadro Interrogazioni, aggiungi la seguente query:

```
query allPostsByTag {  
  allPostsByTag(tag: "dog") {  
    posts {  
      id  
      title  
      tags  
    }  
    nextToken  
  }  
}
```

13 Scegli Esegui (il pulsante di riproduzione arancione), quindi scegli `allPostsByTag`. Vengono restituiti tutti i post con il tag `dog` come segue:

```
{  
  "data": {  
    "allPostsByTag": {  
      "posts": [  
        {  
          "id": "10",  
          "title": "The cutest dog in the world",  
          "tags": [  
            "dog",  
            "puppy"  
          ]  
        }  
      ],  
      "nextToken": null  
    }  
  }  
}
```

```
}
```

Conclusioni

In questo tutorial, hai creato un'API che ti consente di manipolare Post oggetti in DynamoDB AWS AppSync utilizzando GraphQL.

Per eseguire la pulizia, puoi eliminare l'API AWS AppSync GraphQL dalla console.

Per eliminare il ruolo associato alla tabella DynamoDB, seleziona l'origine dati nella tabella Fonti dati e fai clic su modifica. Annota il valore del ruolo in Crea o usa un ruolo esistente. Vai alla console IAM per eliminare il ruolo.

Per eliminare la tabella DynamoDB, fai clic sul nome della tabella nell'elenco delle fonti di dati. Si accede alla console DynamoDB dove è possibile eliminare la tabella.

Utilizzo dei AWS Lambda resolver in AWS AppSync

Puoi usare AWS Lambda with AWS AppSync per risolvere qualsiasi campo GraphQL. Ad esempio, una query GraphQL potrebbe inviare una chiamata a un'istanza Amazon Relational Database Service (Amazon RDS) e una mutazione GraphQL potrebbe scrivere su un flusso Amazon Kinesis. In questa sezione, ti mostreremo come scrivere una funzione Lambda che esegue la logica aziendale basata sull'invocazione di un'operazione sul campo GraphQL.

Powertools per AWS Lambda

Il gestore di eventi Powertools for AWS Lambda GraphQL semplifica il routing e l'elaborazione degli eventi GraphQL nelle funzioni Lambda. È disponibile per Python e Typescript. Per saperne di più sul gestore di eventi dell'API GraphQL, consulta Powertools per la AWS Lambda documentazione, consulta i seguenti riferimenti.

- [Powertools per il gestore di eventi AWS Lambda GraphQL \(Python\)](#)
- [Powertools per il gestore di eventi AWS Lambda GraphQL \(Typescript\)](#)

Creazione di una funzione Lambda

L'esempio seguente mostra una funzione Lambda scritta in Node.js (runtime: Node.js 18.x) che esegue diverse operazioni sui post del blog come parte di un'applicazione per post di blog. Nota che il codice deve essere salvato in un nome di file con estensione.mis.

```
export const handler = async (event) => {
  console.log('Received event {}'.format(JSON.stringify(event, 3)))

  const posts = {
    1: { id: '1', title: 'First book', author: 'Author1', url: 'https://amazon.com/',
      content: 'SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT
AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1', ups: '100', downs: '10', },
    2: { id: '2', title: 'Second book', author: 'Author2', url: 'https://amazon.com',
      content: 'SAMPLE TEXT AUTHOR 2 SAMPLE TEXT AUTHOR 2 SAMPLE TEXT', ups: '100', downs:
'10', },
    3: { id: '3', title: 'Third book', author: 'Author3', url: null, content: null,
      ups: null, downs: null },
    4: { id: '4', title: 'Fourth book', author: 'Author4', url: 'https://
www.amazon.com/', content: 'SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
AUTHOR 4 SAMPLE TEXT AUTHOR 4', ups: '1000', downs: '0', },
    5: { id: '5', title: 'Fifth book', author: 'Author5', url: 'https://
www.amazon.com/', content: 'SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT
AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT', ups: '50', downs: '0', },
  }

  const relatedPosts = {
    1: [posts['4']],
    2: [posts['3'], posts['5']],
    3: [posts['2'], posts['1']],
    4: [posts['2'], posts['1']],
    5: [],
  }

  console.log('Got an Invoke Request.')
  let result
  switch (event.field) {
case 'getPost':
  return posts[event.arguments.id]
case 'allPosts':
  return Object.values(posts)
case 'addPost':
```

```
// return the arguments back
return event.arguments
  case 'addPostErrorWithData':
    result = posts[event.arguments.id]
    // attached additional error information to the post
    result.errorMessage = 'Error with the mutation, data has changed'
    result.errorType = 'MUTATION_ERROR'
  return result
  case 'relatedPosts':
    return relatedPosts[event.source.id]
  default:
    throw new Error('Unknown field, unable to resolve ' + event.field)
}
```

Questa funzione Lambda recupera un post per ID, aggiunge un post, recupera un elenco di post e recupera i post correlati per un determinato post.

Note

La funzione Lambda utilizza l'istruzione `on event.field` per determinare quale campo è attualmente in fase di risoluzione.

Crea questa funzione Lambda utilizzando la console di AWS gestione.

Configurare un'origine dati per Lambda

Dopo aver creato la funzione Lambda, accedi all'API GraphQL nella AWS AppSync console, quindi scegli la scheda Data Sources.

Scegli Crea origine dati, inserisci un nome di origine dati descrittivo (ad esempio **Lambda**), quindi per Tipo di origine dati, scegli AWS Lambda funzione. Per Regione, scegli la stessa regione della tua funzione. Per Function ARN, scegli l'Amazon Resource Name (ARN) della tua funzione Lambda.

Dopo aver scelto la funzione Lambda, puoi creare un nuovo ruolo AWS Identity and Access Management (IAM) (per il quale AWS AppSync assegna le autorizzazioni appropriate) o scegliere un ruolo esistente con la seguente politica in linea:

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "lambda:InvokeFunction"
      ],
      "Resource": "arn:aws:lambda:us-east-1:111122223333:function:LAMBDA_FUNCTION"
    }
  ]
}
```

È inoltre necessario impostare una relazione di fiducia con AWS AppSync per il ruolo IAM nel modo seguente:

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "appsync.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

Creare uno schema GraphQL

Ora che l'origine dati è connessa alla funzione Lambda, crea uno schema GraphQL.

Dall'editor di schemi nella AWS AppSync console, assicurati che lo schema corrisponda allo schema seguente:

```
schema {  
  query: Query  
  mutation: Mutation  
}  
type Query {  
  getPost(id:ID!): Post  
  allPosts: [Post]  
}  
type Mutation {  
  addPost(id: ID!, author: String!, title: String, content: String, url: String):  
    Post!  
}  
type Post {  
  id: ID!  
  author: String!  
  title: String  
  content: String  
  url: String  
  ups: Int  
  downs: Int  
  relatedPosts: [Post]  
}
```

Configura i resolver

Ora che hai registrato un'origine dati Lambda e uno schema GraphQL valido, puoi connettere i campi GraphQL all'origine dati Lambda utilizzando i resolver.

Creerai un resolver che utilizza il runtime AWS AppSync JavaScript (APPSYNC_JS) e interagirà con le tue funzioni Lambda. Per ulteriori informazioni sulla scrittura di AWS AppSync resolver e funzioni con JavaScript, consulta le funzionalità di [JavaScript runtime](#) per resolver e funzioni.

Per ulteriori informazioni sui modelli di mappatura Lambda, consulta il riferimento alla [JavaScript funzione resolver](#) per Lambda.

In questo passaggio, si collega un resolver alla funzione Lambda per i seguenti campi: `getPost(id:ID!): Post`, `allPosts: [Post]` e `addPost(id: ID!, author: String!, title: String, content: String, url: String): Post!`

`Post.relatedPosts: [Post]` Dall'editor di schema della AWS AppSync console, nel riquadro Resolver, scegli **Allega** accanto al campo `getPost(id:ID!): Post` Scegli la tua fonte di dati Lambda. Quindi, fornisci il seguente codice:

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const {source, args} = ctx
  return {
    operation: 'Invoke',
    payload: { field: ctx.info.fieldName, arguments: args, source },
  };
}

export function response(ctx) {
  return ctx.result;
}
```

Questo codice resolver passa il nome del campo, l'elenco di argomenti e il contesto relativo all'oggetto sorgente alla funzione Lambda quando viene richiamata. Scegli **Save** (Salva).

Il primo resolver è stato ora collegato con successo. Ripetere questa operazione per i campi rimanenti.

Testa la tua API GraphQL

Ora che la funzione Lambda è connessa ai resolver GraphQL, puoi eseguire alcune mutazioni e query usando la console o un'applicazione client.

Sul lato sinistro della AWS AppSync console, scegli **Queries**, quindi incolla il codice seguente:

addPost Mutation

```
mutation AddPost {
  addPost(
    id: 6
    author: "Author6"
    title: "Sixth book"
    url: "https://www.amazon.com/"
    content: "This is the book is a tutorial for using GraphQL with AWS AppSync."
  ) {
```

```
    id
    author
    title
    content
    url
    ups
    downs
  }
}
```

getPost Query

```
query GetPost {
  getPost(id: "2") {
    id
    author
    title
    content
    url
    ups
    downs
  }
}
```

allPosts Query

```
query AllPosts {
  allPosts {
    id
    author
    title
    content
    url
    ups
    downs
    relatedPosts {
      id
      title
    }
  }
}
```

Errori di restituzione

Qualsiasi risoluzione di campo può causare un errore. Con AWS AppSync, puoi generare errori dalle seguenti fonti:

- Gestore di risposte Resolver
- funzione Lambda

Dal gestore di risposte del resolver

Per generare errori intenzionali, è possibile utilizzare il metodo di utilità `util.error`. Richiede un argomento `errorMessage`, un `errorType`, un `data` valore opzionale. Il valore `data` è utile per restituire dati aggiuntivi al client quando è stato generato un errore. L'oggetto `data` verrà aggiunto a `errors` nella risposta finale di GraphQL.

L'esempio seguente mostra come usarlo nel gestore di risposte del `Post.relatedPosts`: [Post] resolver.

```
// the Post.relatedPosts response handler
export function response(ctx) {
  util.error("Failed to fetch relatedPosts", "LambdaFailure", ctx.result)
  return ctx.result;
}
```

Ciò restituirà una risposta di GraphQL simile alla seguente:

```
{
  "data": {
    "allPosts": [
      {
        "id": "2",
        "title": "Second book",
        "relatedPosts": null
      },
      ...
    ]
  },
  "errors": [
    {
      "path": [
```

```

        "allPosts",
        0,
        "relatedPosts"
    ],
    "errorType": "LambdaFailure",
    "locations": [
        {
            "line": 5,
            "column": 5
        }
    ],
    "message": "Failed to fetch relatedPosts",
    "data": [
        {
            "id": "2",
            "title": "Second book"
        },
        {
            "id": "1",
            "title": "First book"
        }
    ]
}
]
}

```

Dove `allPosts[0].relatedPosts` è null a causa dell'errore e `errorMessage`, `errorType` e `data` sono presenti nell'oggetto `data.errors[0]`.

Dalla funzione Lambda

AWS AppSync comprende anche gli errori generati dalla funzione Lambda. Il modello di programmazione Lambda consente di generare errori gestiti. Se la funzione Lambda genera un errore, AWS AppSync non riesce a risolvere il campo corrente. Nella risposta viene impostato solo il messaggio di errore restituito da Lambda. Attualmente, non è possibile restituire dati estranei al client generando un errore dalla funzione Lambda.

Note

Se la funzione Lambda genera un errore non gestito, AWS AppSync utilizza il messaggio di errore impostato da Lambda.

La funzione Lambda seguente genera un errore:

```
export const handler = async (event) => {
  console.log('Received event {}'.format(JSON.stringify(event, 3)))
  throw new Error('I always fail.')
}
```

L'errore viene ricevuto nel gestore delle risposte. Puoi rispedirlo nella risposta GraphQL aggiungendo l'errore alla risposta con `util.appendError`. Per fare ciò, modifica il gestore AWS AppSync della risposta alla funzione in questo modo:

```
// the lambdaInvoke response handler
export function response(ctx) {
  const { error, result } = ctx;
  if (error) {
    util.appendError(error.message, error.type, result);
  }
  return result;
}
```

Ciò restituirà una risposta di GraphQL simile alla seguente:

```
{
  "data": {
    "allPosts": null
  },
  "errors": [
    {
      "path": [
        "allPosts"
      ],
      "data": null,
      "errorType": "Lambda:Unhandled",
      "errorInfo": null,
      "locations": [
        {
          "line": 2,
          "column": 3,
          "sourceName": null
        }
      ],
      "message": "I fail. always"
    }
  ]
}
```

```
    }  
  ]  
}
```

Caso d'uso avanzato: Batching

La funzione Lambda in questo esempio ha un `relatedPosts` campo che restituisce un elenco di post correlati per un determinato post. Nelle query di esempio, l'invocazione del `allPosts` campo dalla funzione Lambda restituisce cinque post. Poiché abbiamo specificato che vogliamo risolvere anche `relatedPosts` per ogni post restituito, l'operazione `relatedPosts` sul campo viene richiamata cinque volte.

```
query {  
  allPosts {    // 1 Lambda invocation - yields 5 Posts  
    id  
    author  
    title  
    content  
    url  
    ups  
    downs  
    relatedPosts {    // 5 Lambda invocations - each yields 5 posts  
      id  
      title  
    }  
  }  
}
```

Anche se questo potrebbe non sembrare importante in questo esempio specifico, questo eccesso di recupero aggravato può compromettere rapidamente l'applicazione.

Se, ad esempio, dovessimo recuperare di nuovo `relatedPosts` sui `Posts` correlati restituiti nella stessa query, il numero di chiamate aumenterebbe notevolmente.

```
query {  
  allPosts {    // 1 Lambda invocation - yields 5 Posts  
    id  
    author  
    title  
    content  
    url
```



```

    ups
    downs
    relatedPosts { // 5 Lambda invocations - each yield 5 posts = 5 x 5 Posts
      id
      title
      relatedPosts { // 5 x 5 Lambda invocations - each yield 5 posts = 25 x 5
        id
        title
        author
      }
    }
  }
}

```

In questa query relativamente semplice, AWS AppSync richiamerebbe la funzione Lambda $1 + 5 + 25 = 31$ volte.

Si tratta di una sfida piuttosto comune a cui si fa spesso riferimento come problema N+1, (nel nostro caso, $N = 5$) che può determinare un aumento della latenza e dei costi dell'applicazione.

Un approccio alla soluzione di questo problema consiste nel raggruppare in batch richieste del resolver di campo simili. In questo esempio, invece di fare in modo che la funzione Lambda risolva un elenco di post correlati per un singolo post, potrebbe invece risolvere un elenco di post correlati per un determinato batch di post.

Per dimostrarlo, aggiorniamo il resolver per `relatedPosts` gestire il batching.

```

import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const {source, args} = ctx
  return {
    operation: ctx.info.fieldName === 'relatedPosts' ? 'BatchInvoke' : 'Invoke',
    payload: { field: ctx.info.fieldName, arguments: args, source },
  };
}

export function response(ctx) {
  const { error, result } = ctx;
  if (error) {
    util.appendError(error.message, error.type, result);
  }
}

```

```
}  
  return result;  
}
```

Il codice ora modifica l'operazione da `Invoke` a `BatchInvoke` quando è in `fieldName` corso la risoluzione. `relatedPosts` Ora, abilita il batching sulla funzione nella sezione Configura batch. Imposta la dimensione massima di batch impostata su. 5 Scegli **Save (Salva)**.

Con questa modifica, durante la risoluzione `relatedPosts`, la funzione Lambda riceve quanto segue come input:

```
[  
  {  
    "field": "relatedPosts",  
    "source": {  
      "id": 1  
    }  
  },  
  {  
    "field": "relatedPosts",  
    "source": {  
      "id": 2  
    }  
  },  
  ...  
]
```

Quando `BatchInvoke` è specificata nella richiesta, la funzione Lambda riceve un elenco di richieste e restituisce un elenco di risultati.

In particolare, l'elenco dei risultati deve corrispondere alla dimensione e all'ordine delle voci del payload della richiesta in modo che AWS AppSync possa corrispondere ai risultati di conseguenza.

In questo esempio di batch, la funzione Lambda restituisce un batch di risultati come segue:

```
[  
  [{ "id": "2", "title": "Second book" }, { "id": "3", "title": "Third book" } ], //  
  relatedPosts for id=1  
  [{ "id": "3", "title": "Third book" } ] //  
  relatedPosts for id=2  
]
```

Puoi aggiornare il codice Lambda per gestire il batch per: relatedPosts

```
export const handler = async (event) => {
  console.log('Received event {}'.format(JSON.stringify(event, 3)))
  //throw new Error('I fail. always')

  const posts = {
    1: { id: '1', title: 'First book', author: 'Author1', url: 'https://amazon.com/',
      content: 'SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT
      AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1', ups: '100', downs: '10', },
    2: { id: '2', title: 'Second book', author: 'Author2', url: 'https://amazon.com',
      content: 'SAMPLE TEXT AUTHOR 2 SAMPLE TEXT AUTHOR 2 SAMPLE TEXT', ups: '100', downs:
      '10', },
    3: { id: '3', title: 'Third book', author: 'Author3', url: null, content: null,
      ups: null, downs: null },
    4: { id: '4', title: 'Fourth book', author: 'Author4', url: 'https://
      www.amazon.com/', content: 'SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
      AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
      AUTHOR 4 SAMPLE TEXT AUTHOR 4', ups: '1000', downs: '0', },
    5: { id: '5', title: 'Fifth book', author: 'Author5', url: 'https://
      www.amazon.com/', content: 'SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT
      AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT', ups: '50', downs: '0', },
  }

  const relatedPosts = {
    1: [posts['4']],
    2: [posts['3'], posts['5']],
    3: [posts['2'], posts['1']],
    4: [posts['2'], posts['1']],
    5: [],
  }

  if (!event.field && event.length){
    console.log(`Got a BatchInvoke Request. The payload has ${event.length} items to
    resolve.`);
    return event.map(e => relatedPosts[e.source.id])
  }

  console.log('Got an Invoke Request.')
  let result
  switch (event.field) {
    case 'getPost':
      return posts[event.arguments.id]
    case 'allPosts':
```

```

    return Object.values(posts)
  case 'addPost':
    // return the arguments back
    return event.arguments
  case 'addPostErrorWithData':
    result = posts[event.arguments.id]
    // attached additional error information to the post
    result.errorMessage = 'Error with the mutation, data has changed'
    result.errorType = 'MUTATION_ERROR'
    return result
  case 'relatedPosts':
    return relatedPosts[event.source.id]
  default:
    throw new Error('Unknown field, unable to resolve ' + event.field)
}
}

```

Restituzione di errori individuali

Gli esempi precedenti mostrano che è possibile restituire un singolo errore dalla funzione Lambda o generare un errore dal gestore delle risposte. Per le chiamate in batch, la generazione di un errore dalla funzione Lambda contrassegna un intero batch come fallito. Questo potrebbe essere accettabile per scenari specifici in cui si verifica un errore irreversibile, ad esempio una connessione non riuscita a un data store. Tuttavia, nei casi in cui alcuni elementi del batch abbiano esito positivo e altri falliscano, è possibile restituire sia errori che dati validi. Poiché AWS AppSync richiede la risposta in batch agli elementi dell'elenco che corrispondono alla dimensione originale del batch, è necessario definire una struttura di dati in grado di differenziare i dati validi da un errore.

Ad esempio, se si prevede che la funzione Lambda restituisca un batch di post correlati, è possibile scegliere di restituire un elenco di *Response* oggetti in cui ogni oggetto contiene dati opzionali, campi `ErrorMessage` ed `ErrorType`. La presenza del campo `errorMessage` indica un errore.

Il codice seguente mostra come aggiornare la funzione Lambda:

```

export const handler = async (event) => {
  console.log('Received event {}', JSON.stringify(event, 3))
  // throw new Error('I fail. always')
  const posts = {
    1: { id: '1', title: 'First book', author: 'Author1', url: 'https://amazon.com/',
      content: 'SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1', ups: '100', downs: '10', },
  }
}

```

```

    2: { id: '2', title: 'Second book', author: 'Author2', url: 'https://amazon.com',
content: 'SAMPLE TEXT AUTHOR 2 SAMPLE TEXT AUTHOR 2 SAMPLE TEXT', ups: '100', downs:
'10', },
    3: { id: '3', title: 'Third book', author: 'Author3', url: null, content: null,
ups: null, downs: null },
    4: { id: '4', title: 'Fourth book', author: 'Author4', url: 'https://
www.amazon.com/', content: 'SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
AUTHOR 4 SAMPLE TEXT AUTHOR 4', ups: '1000', downs: '0', },
    5: { id: '5', title: 'Fifth book', author: 'Author5', url: 'https://
www.amazon.com/', content: 'SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT
AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT', ups: '50', downs: '0', },
  }

  const relatedPosts = {
1: [posts['4']],
    2: [posts['3'], posts['5']],
    3: [posts['2'], posts['1']],
    4: [posts['2'], posts['1']],
    5: [],
  }

  if (!event.field && event.length){
console.log(`Got a BatchInvoke Request. The payload has ${event.length} items to
resolve.`);
    return event.map(e => {
// return an error for post 2
if (e.source.id === '2') {
return { 'data': null, 'errorMessage': 'Error Happened', 'errorType': 'ERROR' }
}
    return {data: relatedPosts[e.source.id]}
  })
  }

  console.log('Got an Invoke Request.')
  let result
  switch (event.field) {
case 'getPost':
    return posts[event.arguments.id]
case 'allPosts':
    return Object.values(posts)
case 'addPost':
    // return the arguments back
return event.arguments

```

```

    case 'addPostErrorWithData':
      result = posts[event.arguments.id]
      // attached additional error information to the post
      result.errorMessage = 'Error with the mutation, data has changed'
      result.errorType = 'MUTATION_ERROR'
    return result
    case 'relatedPosts':
      return relatedPosts[event.source.id]
    default:
      throw new Error('Unknown field, unable to resolve ' + event.field)
  }
}

```

Aggiorna il codice del `relatedPosts` resolver:

```

import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const {source, args} = ctx
  return {
    operation: ctx.info.fieldName === 'relatedPosts' ? 'BatchInvoke' : 'Invoke',
    payload: { field: ctx.info.fieldName, arguments: args, source },
  };
}

export function response(ctx) {
  const { error, result } = ctx;
  if (error) {
    util.appendError(error.message, error.type, result);
  } else if (result.errorMessage) {
    util.appendError(result.errorMessage, result.errorType, result.data)
  } else if (ctx.info.fieldName === 'relatedPosts') {
    return result.data
  } else {
    return result
  }
}

```

Il gestore delle risposte ora controlla gli errori restituiti dalla funzione `Lambda Invoke` sulle operazioni, controlla gli errori restituiti per i singoli elementi `BatchInvoke` per le operazioni e infine controlla il `fieldName`. Perché `relatedPosts`, la funzione restituisce `result.data`. Per tutti gli altri campi, la funzione si limita a restituire `result`. Ad esempio, vedi la query seguente:

```
query AllPosts {  
  allPosts {  
    id  
    title  
    content  
    url  
    ups  
    downs  
    relatedPosts {  
      id  
    }  
    author  
  }  
}
```

Questa query restituisce una risposta GraphQL simile alla seguente:

```
{  
  "data": {  
    "allPosts": [  
      {  
        "id": "1",  
        "relatedPosts": [  
          {  
            "id": "4"  
          }  
        ]  
      },  
      {  
        "id": "2",  
        "relatedPosts": null  
      },  
      {  
        "id": "3",  
        "relatedPosts": [  
          {  
            "id": "2"  
          },  
          {  
            "id": "1"  
          }  
        ]  
      }  
    ]  
  },  
}
```

```
{
  {
    "id": "4",
    "relatedPosts": [
      {
        "id": "2"
      },
      {
        "id": "1"
      }
    ]
  },
  {
    "id": "5",
    "relatedPosts": []
  }
],
"errors": [
  {
    "path": [
      "allPosts",
      1,
      "relatedPosts"
    ],
    "data": null,
    "errorType": "ERROR",
    "errorInfo": null,
    "locations": [
      {
        "line": 4,
        "column": 5,
        "sourceName": null
      }
    ],
    "message": "Error Happened"
  }
]
```

Configurazione della dimensione massima di batch

Per configurare la dimensione massima di batch su un resolver, utilizzate il seguente comando in ():
AWS Command Line Interface AWS CLI


```
$ aws appsync create-resolver --api-id <api-id> --type-name Query --field-name
relatedPosts \
--code "<code-goes-here>" \
--runtime name=APPSYNC_JS,runtimeVersion=1.0.0 \
--data-source-name "<lambda-datasource>" \
--max-batch-size X
```

Note

Quando si fornisce un modello di mappatura delle richieste, è necessario utilizzare l'BatchInvokeoperazione per utilizzare il batch.

Utilizzo di resolver locali in AWS AppSync

AWS AppSync consente di utilizzare fonti di dati supportate (AWS Lambda Amazon DynamoDB o OpenSearch Amazon Service) per eseguire varie operazioni. In alcuni scenari, tuttavia, una chiamata a un'origine dati supportata può non essere necessaria.

È in queste situazioni che il resolver locale risulta utile. Invece di chiamare una fonte di dati remota, il resolver locale inoltrerà semplicemente il risultato del gestore della richiesta al gestore della risposta. La risoluzione del campo rimarrà in AWS AppSync.

I resolver locali sono utili in una miriade di situazioni. Quello più comune è la pubblicazione di notifiche senza attivare una chiamata a un'origine dati. Per dimostrare questo caso d'uso, creiamo un'applicazione pub/sub in cui gli utenti possano pubblicare e iscriversi ai messaggi. In questo esempio vengono usate le sottoscrizioni. Quindi, se non hai familiarità con le sottoscrizioni, puoi seguire il tutorial [Dati in tempo reale](#).

Creazione dell'app pub/sub

Innanzitutto, crea un'API GraphQL vuota scegliendo l'opzione Design from scratch e configurando i dettagli opzionali durante la creazione dell'API GraphQL.

Nella nostra applicazione pub/sub, i clienti possono iscriversi e pubblicare messaggi. Ogni messaggio pubblicato include un nome e dei dati. Aggiungi questo allo schema:

```
type Channel {
  name: String!
  data: AWSJSON!
```

```
}

type Mutation {
  publish(name: String!, data: AWSJSON!): Channel!
}

type Query {
  getChannel: Channel!
}

type Subscription {
  subscribe(name: String!): Channel!
  @aws_subscribe(mutations: ["publish"])
}
```

Quindi, colleghiamo un resolver al `Mutation.publish` campo. Nel riquadro Resolver accanto al riquadro Schema, trova il **Mutation** tipo, quindi il **publish(...): Channel** campo, quindi fai clic su **Allega**.

Crea una fonte di dati Nessuno e assegna un nome. `PageDataSource` Collegalo al tuo resolver.

Aggiungi l'implementazione del tuo resolver utilizzando il seguente frammento:

```
export function request(ctx) {
  return { payload: ctx.args };
}

export function response(ctx) {
  return ctx.result;
}
```

Assicurati di creare il resolver e di salvare le modifiche apportate.

Invia e sottoscrivi messaggi

Affinché i clienti possano ricevere messaggi, devono prima essere iscritti a una casella di posta.

Nel riquadro Query, esegui l'abbonamento: `SubscribeToData`

```
subscription SubscribeToData {
  subscribe(name:"channel") {
    name
    data
  }
}
```

```
}  
}
```

Il sottoscrittore riceverà messaggi ogni volta che viene richiamata la `publish` mutazione, ma solo quando il messaggio viene inviato all'abbonamento. `channel` Proviamo a farlo nel riquadro Query. Mentre l'abbonamento è ancora in esecuzione nella console, apri un'altra console ed esegui la seguente richiesta nel riquadro Query:

Note

In questo esempio utilizziamo stringhe JSON valide.

```
mutation PublishData {  
  publish(data: "{\"msg\": \"hello world!\"}", name: "channel") {  
    data  
    name  
  }  
}
```

Il risultato sarà simile al seguente:

```
{  
  "data": {  
    "publish": {  
      "data": "{\"msg\": \"hello world!\"}",  
      "name": "channel"  
    }  
  }  
}
```

Abbiamo appena dimostrato l'uso dei resolver locali, pubblicando un messaggio e ricevendolo senza uscire dal servizio. AWS AppSync

Combinazione di resolver GraphQL in AWS AppSync

I resolver e i campi in uno schema di GraphQL hanno un rapporto 1:1 con un elevato grado di flessibilità. Poiché un'origine dati è configurata su un resolver indipendentemente da uno schema, hai la possibilità di risolvere o manipolare i tuoi tipi di GraphQL attraverso diverse fonti di dati, consentendoti di combinare uno schema per soddisfare al meglio le tue esigenze.

Gli scenari seguenti mostrano come combinare e abbinare le fonti di dati nello schema. Prima di iniziare, dovresti avere familiarità con la configurazione di sorgenti di dati e resolver per Amazon AWS Lambda DynamoDB e Amazon Service. OpenSearch

Schema di esempio

Lo schema seguente ha un tipo di Mutation operazioni Post con tre Query e ciascuna:

```
type Post {
  id: ID!
  author: String!
  title: String
  content: String
  url: String
  ups: Int
  downs: Int
  version: Int!
}

type Query {
  allPost: [Post]
  getPost(id: ID!): Post
  searchPosts: [Post]
}

type Mutation {
  addPost(
    id: ID!,
    author: String!,
    title: String,
    content: String,
    url: String
  ): Post
  updatePost(
    id: ID!,
    author: String!,
    title: String,
    content: String,
    url: String,
    ups: Int!,
    downs: Int!,
    expectedVersion: Int!
  ): Post
}
```

```
deletePost(id: ID!): Post
}
```

In questo esempio, avresti un totale di sei resolver, ognuno dei quali necessita di una fonte di dati. Un modo per risolvere questo problema sarebbe collegarli a una singola tabella Amazon DynamoDB, Posts chiamata, in cui AllPost il campo esegue una scansione e il campo esegue una query ([JavaScriptvedi riferimento searchPosts alla funzione resolver](#) per DynamoDB). Tuttavia, non sei limitato ad Amazon DynamoDB; esistono diverse fonti di dati come Lambda OpenSearch o Service per soddisfare le tue esigenze aziendali.

Alterazione dei dati tramite resolver

Potrebbe essere necessario restituire i risultati da un database di terze parti non supportato direttamente dalle fonti di dati. AWS AppSync Potrebbe inoltre essere necessario eseguire modifiche complesse sui dati prima che vengano restituiti ai client API. Ciò potrebbe essere causato da una formattazione impropria dei tipi di dati, ad esempio differenze di timestamp sui client o dalla gestione di problemi di compatibilità con le versioni precedenti. In questo caso, la soluzione appropriata è connettere AWS Lambda le funzioni come fonte di dati all' AWS AppSync API. A scopo illustrativo, nell'esempio seguente, una AWS Lambda funzione manipola i dati recuperati da un archivio dati di terze parti:

```
export const handler = (event, context, callback) => {
  // fetch data
  const result = fetcher()

  // apply complex business logic
  const data = transform(result)

  // return to AppSync
  return data
};
```

Questa è una funzione di Lambda perfettamente valida e potrebbe essere associata al campo AllPost nello schema di GraphQL in modo che qualsiasi query che restituisce tutti i risultati ottenga numeri casuali per i voti in alto/in basso.

DynamoDB e Service OpenSearch

Per alcune applicazioni, è possibile eseguire mutazioni o semplici query di ricerca su DynamoDB e disporre di un processo in background per trasferire i documenti al Service. OpenSearch È sufficiente

collegare il `searchPosts` resolver all'origine dati del OpenSearch servizio e restituire i risultati della ricerca (dai dati originati in DynamoDB) utilizzando una query GraphQL. Ciò può rivelarsi estremamente efficace quando si aggiungono operazioni di ricerca avanzate alle applicazioni, ad esempio parole chiave, corrispondenze di parole confuse o persino ricerche geospaziali. Il trasferimento di dati da DynamoDB può essere eseguito tramite un processo ETL o, in alternativa, è possibile eseguire lo streaming da DynamoDB utilizzando Lambda.

Per iniziare con queste particolari fonti di dati, consulta i nostri tutorial su [DynamoDB e Lambda](#).

Ad esempio, utilizzando lo schema del nostro tutorial precedente, la seguente mutazione aggiunge un elemento a DynamoDB:

```
mutation addPost {  
  addPost(  
    id: 123  
    author: "Nadia"  
    title: "Our first post!"  
    content: "This is our first post."  
    url: "https://aws.amazon.com/appsync/"  
  ) {  
    id  
    author  
    title  
    content  
    url  
    ups  
    downs  
    version  
  }  
}
```

Questo scrive i dati su DynamoDB, che quindi trasmette i dati tramite Lambda ad OpenSearch Amazon Service, che poi usi per cercare post in base a campi diversi. Ad esempio, poiché i dati si trovano in Amazon OpenSearch Service, puoi cercare nei campi dell'autore o del contenuto con testo in formato libero, anche con spazi, come segue:

```
query searchName{  
  searchAuthor(name:"  Nadia  "){  
    id  
    title  
    content  
  }  
}
```

```

}

----- or -----

query searchContent{
  searchContent(text:"test"){
    id
    title
    content
  }
}

```

Poiché i dati vengono scritti direttamente su DynamoDB, è comunque possibile eseguire operazioni efficienti di ricerca di elenchi o elementi sulla tabella con le query `and`. `allPost{...}` `getPost{...}` Questo stack utilizza il seguente codice di esempio per i flussi DynamoDB:

Note

Questo codice Python è un esempio e non è pensato per essere utilizzato nel codice di produzione.

```

import boto3
import requests
from requests_aws4auth import AWS4Auth

region = '' # e.g. us-east-1
service = 'es'
credentials = boto3.Session().get_credentials()
awsauth = AWS4Auth(credentials.access_key, credentials.secret_key, region, service,
    session_token=credentials.token)

host = '' # the OpenSearch Service domain, e.g. https://search-mydomain.us-
west-1.es.amazonaws.com
index = 'lambda-index'
datatype = '_doc'
url = host + '/' + index + '/' + datatype + '/'

headers = { "Content-Type": "application/json" }

def handler(event, context):
    count = 0

```

```
for record in event['Records']:
    # Get the primary key for use as the OpenSearch ID
    id = record['dynamodb']['Keys']['id']['S']

    if record['eventName'] == 'REMOVE':
        r = requests.delete(url + id, auth=awsauth)
    else:
        document = record['dynamodb']['NewImage']
        r = requests.put(url + id, auth=awsauth, json=document, headers=headers)
    count += 1
return str(count) + ' records processed.'
```

Puoi quindi utilizzare i flussi DynamoDB per collegarla a una tabella DynamoDB con una chiave primaria di e qualsiasi modifica all'origine id di DynamoDB verrà trasferita nel tuo dominio di servizio. OpenSearch Per ulteriori informazioni sulla configurazione di questa funzionalità, consulta [Documentazione dei flussi di DynamoDB](#).

Utilizzo dei OpenSearch resolver Amazon Service in AWS AppSync

AWS AppSync supporta l'utilizzo di Amazon OpenSearch Service da domini che hai fornito nel tuo AWS account, a condizione che non esistano all'interno di un VPC. Dopo che sono stati assegnati i domini, è possibile connettersi a essi tramite un'origine dati, a quel punto è possibile configurare un resolver nello schema per eseguire operazioni di GraphQL, come ad esempio query, mutazioni e iscrizioni. Questo tutorial fornirà una descrizione di alcuni esempi comuni.

Per ulteriori informazioni, consulta la nostra guida di riferimento alla funzione [JavaScript resolver per OpenSearch](#)

Crea un nuovo dominio di servizio OpenSearch

Per iniziare con questo tutorial, è necessario un dominio di OpenSearch servizio esistente. Se non si dispone di un dominio, è possibile usare il campione seguente. Tieni presente che possono essere necessari fino a 15 minuti per la creazione di un dominio di OpenSearch servizio prima di poter passare all'integrazione con una fonte di AWS AppSync dati.

```
aws cloudformation create-stack --stack-name AppSyncOpenSearch \
--template-url https://s3.us-west-2.amazonaws.com/awsappsync/resources/elasticsearch/
ESResolverCFTemplate.yaml \
--parameters ParameterKey=OSDomainName,ParameterValue=ddtestdomain
ParameterKey=Tier,ParameterValue=development \
```



```
--capabilities CAPABILITY_NAMED_IAM
```

Puoi avviare il seguente AWS CloudFormation stack nella regione US-West-2 (Oregon) nel tuo account: AWS

Launch Stack



Configura una fonte di dati per Service OpenSearch

Dopo aver creato il dominio OpenSearch Service, accedi all'API AWS AppSync GraphQL e scegli la scheda Data Sources. Scegli Crea origine dati e inserisci un nome descrittivo per l'origine dati, ad esempio «oss». Quindi, scegli il OpenSearch dominio Amazon per il tipo di origine dati, scegli la regione appropriata e dovresti vedere il tuo dominio di OpenSearch servizio elencato. Dopo averlo selezionato, puoi creare un nuovo ruolo e AWS AppSync assegnare le autorizzazioni appropriate al ruolo oppure puoi scegliere un ruolo esistente, con la seguente politica in linea:

Dovrai inoltre stabilire una relazione di fiducia con per quel ruolo: AWS AppSync

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "appsync.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

Inoltre, il dominio OpenSearch Service ha una propria politica di accesso che puoi modificare tramite la console di Amazon OpenSearch Service. È necessario aggiungere una politica simile a quella riportata di seguito con le azioni e le risorse appropriate per il dominio del OpenSearch servizio. Tieni presente che il ruolo Principal sarà il ruolo dell'origine AWS AppSync dati, che può essere trovato nella console IAM se lasci che sia la console a crearlo.

Collegamento di un resolver

Ora che l'origine dati è connessa al tuo dominio OpenSearch Service, puoi connetterla allo schema GraphQL con un resolver, come mostrato nell'esempio seguente:

```
type Query {
  getPost(id: ID!): Post
  allPosts: [Post]
}

type Mutation {
  addPost(id: ID!, author: String, title: String, url: String, ups: Int, downs: Int,
content: String): AWSJSON
}

type Post {
  id: ID!
  author: String
  title: String
  url: String
  ups: Int
  downs: Int
  content: String
}
```

Si noti che vi è un tipo `Post` definito dall'utente con un campo di `id`. Negli esempi seguenti, supponiamo che esista un processo (che può essere automatizzato) per inserire questo tipo nel dominio di OpenSearch servizio, che verrebbe mappato alla radice del percorso in `/post/_doc` cui si `post` trova l'indice. Da questo percorso principale, è possibile eseguire ricerche su singoli documenti, ricerche con caratteri jolly o ricerche su più documenti con `/id/post*` un percorso di. `/post/_search` Ad esempio, se avete un altro tipo chiamato `User`, potete indicizzare i documenti in base a un nuovo indice chiamato `user`, quindi eseguire ricerche con un percorso di. `/user/_search`

Dall'editor dello schema nella AWS AppSync console, modifica lo `Posts` schema precedente per includere una `searchPosts` query:

```
type Query {
  getPost(id: ID!): Post
  allPosts: [Post]
  searchPosts: [Post]
```

```
}
```

Salvare lo schema. Nel riquadro Resolver, trova **searchPosts** e scegli Allega. Scegli la fonte dei dati OpenSearch del servizio e salva il resolver. Aggiorna il codice del tuo resolver utilizzando lo snippet seguente:

```
import { util } from '@aws-appsync/utils'

/**
 * Searches for documents by using an input term
 * @param {import('@aws-appsync/utils').Context} ctx the context
 * @returns {*} the request
 */
export function request(ctx) {
  return {
    operation: 'GET',
    path: `/post/_search`,
    params: { body: { from: 0, size: 50 } },
  }
}

/**
 * Returns the fetched items
 * @param {import('@aws-appsync/utils').Context} ctx the context
 * @returns {*} the result
 */
export function response(ctx) {
  if (ctx.error) {
    util.error(ctx.error.message, ctx.error.type)
  }
  return ctx.result.hits.hits.map((hit) => hit._source)
}
```

Ciò presuppone che lo schema precedente contenga documenti che sono stati indicizzati in Service sotto il campo. OpenSearch post Se strutturi i dati in modo diverso, dovrai aggiornarli di conseguenza.

Modificare le ricerche

Il precedente gestore di richieste del resolver esegue una semplice interrogazione per tutti i record. Se si desidera eseguire la ricerca per un autore specifico Inoltre, supponiamo che tu voglia che

quell'autore sia un argomento definito nella tua query GraphQL. Nell'editor dello schema della AWS AppSync console, aggiungi una `allPostsByAuthor` query:

```
type Query {  
  getPost(id: ID!): Post  
  allPosts: [Post]  
  allPostsByAuthor(author: String!): [Post]  
  searchPosts: [Post]  
}
```

Nel riquadro Resolver, trova **`allPostsByAuthor`** e scegli Allega. Scegli l'origine dati del OpenSearch servizio e utilizza il seguente codice:

```
import { util } from '@aws-appsync/utils'  
  
/**  
 * Searches for documents by `author`  
 * @param {import('@aws-appsync/utils').Context} ctx the context  
 * @returns {*} the request  
 */  
export function request(ctx) {  
  return {  
    operation: 'GET',  
    path: '/post/_search',  
    params: {  
      body: {  
        from: 0,  
        size: 50,  
        query: { match: { author: ctx.args.author } },  
      },  
    },  
  },  
}  
  
/**  
 * Returns the fetched items  
 * @param {import('@aws-appsync/utils').Context} ctx the context  
 * @returns {*} the result  
 */  
export function response(ctx) {  
  if (ctx.error) {  
    util.error(ctx.error.message, ctx.error.type)  
  }  
}
```

```
return ctx.result.hits.hits.map((hit) => hit._source)
}
```

Si noti che body viene popolato con una query del termine per il campo author, che è già passato attraverso il client come argomento. Facoltativamente, puoi utilizzare informazioni precompilate, come testo standard.

Aggiungere dati al servizio OpenSearch

Potresti voler aggiungere dati al tuo dominio di OpenSearch servizio come risultato di una mutazione GraphQL. Si tratta di un meccanismo potente per le ricerche e altri scopi. Poiché puoi utilizzare gli abbonamenti GraphQL per [rendere i tuoi dati in tempo reale](#), può fungere da meccanismo per notificare ai clienti gli aggiornamenti dei dati nel tuo dominio di servizio. OpenSearch

Torna alla pagina Schema nella AWS AppSync console e seleziona Allega per la mutazione. `addPost()` Seleziona nuovamente l'origine dati del OpenSearch servizio e utilizza il codice seguente:

```
import { util } from '@aws-appsync/utils'

/**
 * Searches for documents by `author`
 * @param {import('@aws-appsync/utils').Context} ctx the context
 * @returns {*} the request
 */
export function request(ctx) {
  return {
    operation: 'PUT',
    path: `/post/_doc/${ctx.args.id}`,
    params: { body: ctx.args },
  }
}

/**
 * Returns the inserted post
 * @param {import('@aws-appsync/utils').Context} ctx the context
 * @returns {*} the result
 */
export function response(ctx) {
  if (ctx.error) {
    util.error(ctx.error.message, ctx.error.type)
  }
}
```

```
    return ctx.result
  }
```

Come in precedenza, questo è un esempio di come potrebbero essere strutturati i dati. Se hai nomi o indici di campo diversi, devi aggiornare `path` and `body`. Questo esempio mostra anche come utilizzare `context.arguments`, che può anche essere scritto come `ctx.args`, nel gestore delle richieste.

Recupero di un singolo documento

Infine, se desideri utilizzare la **`getPost(id:ID)`** query nello schema per restituire un singolo documento, trova questa query nell'editor di schemi della AWS AppSync console e scegli **Allega**. Seleziona nuovamente l'origine dati del OpenSearch servizio e utilizza il seguente codice:

```
import { util } from '@aws-appsync/utils'

/**
 * Searches for documents by `author`
 * @param {import('@aws-appsync/utils').Context} ctx the context
 * @returns {*} the request
 */
export function request(ctx) {
  return {
    operation: 'GET',
    path: `/post/_doc/${ctx.args.id}`,
  }
}

/**
 * Returns the post
 * @param {import('@aws-appsync/utils').Context} ctx the context
 * @returns {*} the result
 */
export function response(ctx) {
  if (ctx.error) {
    util.error(ctx.error.message, ctx.error.type)
  }
  return ctx.result._source
}
```

Esegui interrogazioni e mutazioni

Ora dovresti essere in grado di eseguire operazioni GraphQL sul tuo dominio di OpenSearch servizio. Vai alla scheda Query della AWS AppSync console e aggiungi un nuovo record:

```
mutation AddPost {  
  addPost (  
    id:"12345"  
    author: "Fred"  
    title: "My first book"  
    content: "This will be fun to write!"  
    url: "publisher website",  
    ups: 100,  
    downs:20  
  )  
}
```

Vedrai il risultato della mutazione sulla destra. Allo stesso modo, ora puoi eseguire una `searchPosts` query sul tuo dominio OpenSearch di servizio:

```
query search {  
  searchPosts {  
    id  
    title  
    author  
    content  
  }  
}
```

Best practice

- OpenSearch Il servizio deve essere utilizzato per l'interrogazione dei dati, non come database principale. [Potresti voler utilizzare OpenSearch Service insieme ad Amazon DynamoDB, come indicato in Combined GraphQL Resolvers.](#)
- Concedi l'accesso al tuo dominio solo consentendo al ruolo di servizio di accedere al cluster. AWS AppSync
- È possibile iniziare con una soluzione di base in fase di sviluppo, con il cluster dal prezzo più basso, e quindi spostarsi in un cluster più grande con elevata disponibilità quando si passa alla produzione.

Esecuzione di transazioni DynamoDB in AWS AppSync

AWS AppSync supporta l'utilizzo di operazioni di transazione Amazon DynamoDB su una o più tabelle in una singola regione. Le operazioni supportate sono `TransactGetItems` e `TransactWriteItems`. Utilizzando queste funzionalità in AWS AppSync, puoi eseguire attività come:

- Passare un elenco di chiavi in una singola query e restituire i risultati da una tabella
- Leggere i record da una o più tabelle in una singola query
- Scrittura dei record nelle transazioni su una o più tabelle in qualsiasi all-or-nothing modo
- Esecuzione di transazioni quando vengono soddisfatte alcune condizioni

Permissions

Come altri resolver, è necessario creare un'origine dati in AWS AppSync e creare un ruolo o utilizzarne uno esistente. Poiché le operazioni di transazione richiedono autorizzazioni diverse sulle tabelle DynamoDB, è necessario concedere ai ruoli configurati le autorizzazioni per le azioni di lettura o scrittura:

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "dynamodb:DeleteItem",
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:Query",
        "dynamodb:Scan",
        "dynamodb:UpdateItem"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:dynamodb:us-east-1:111122223333:table/TABLENAME",
        "arn:aws:dynamodb:us-east-1:111122223333:table/TABLENAME/*"
      ]
    }
  ]
}
```



```
]
}
```

Note

I ruoli sono legati alle fonti di dati in AWS AppSync e i resolver sui campi vengono richiamati su un'origine dati. Le origini dati configurate per il recupero con DynamoDB hanno solo una tabella specificata per semplificare le configurazioni. Pertanto, quando si esegue un'operazione di transazione su più tabelle in un singolo resolver, che è una delle attività più avanzate, è necessario concedere il ruolo sull'accesso all'origine dati a tutte le tabelle con cui interagirà il resolver. Questo potrebbe essere fatto nel campo **Risorsa** nella policy di IAM indicata in precedenza. La configurazione delle chiamate di transazione rispetto alle tabelle viene eseguita nel codice del resolver, che descriveremo di seguito.

Origine dati

Per semplicità, useremo la stessa origine dati per tutti i resolver usati in questo tutorial.

Avremo due tabelle chiamate `SavingAccounts` e `CheckingAccounts`, entrambe **accountNumber** con la chiave di partizione e una tabella `TransactionHistory` con `transactionId` come chiave di partizione. È possibile utilizzare i comandi CLI riportati di seguito per creare le tabelle. Assicurati di sostituirlo `region` con la tua regione.

Con la CLI

```
aws dynamodb create-table --table-name savingAccounts \
  --attribute-definitions AttributeName=accountNumber,AttributeType=S \
  --key-schema AttributeName=accountNumber,KeyType=HASH \
  --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
  --table-class STANDARD --region region

aws dynamodb create-table --table-name checkingAccounts \
  --attribute-definitions AttributeName=accountNumber,AttributeType=S \
  --key-schema AttributeName=accountNumber,KeyType=HASH \
  --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
  --table-class STANDARD --region region

aws dynamodb create-table --table-name transactionHistory \
  --attribute-definitions AttributeName=transactionId,AttributeType=S \
```

```
--key-schema AttributeName=transactionId,KeyType=HASH \
--provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
--table-class STANDARD --region region
```

Nella AWS AppSync console, in Sorgenti dati, crea una nuova origine dati DynamoDB e assegna un nome. TransactTutorial Seleziona SavingAccounts come tabella (sebbene la tabella specifica non abbia importanza quando si utilizzano le transazioni). Scegli di creare un nuovo ruolo e l'origine dati. Puoi rivedere la configurazione dell'origine dati per vedere il nome del ruolo generato. Nella console IAM, puoi aggiungere una policy in linea che consente all'origine dati di interagire con tutte le tabelle.

Sostituisci `region` e `accountID` con la tua regione e l'ID dell'account:

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "dynamodb:DeleteItem",
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:Query",
        "dynamodb:Scan",
        "dynamodb:UpdateItem"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:dynamodb:us-east-1:111122223333:table/savingAccounts",
        "arn:aws:dynamodb:us-east-1:111122223333:table/savingAccounts/*",
        "arn:aws:dynamodb:us-east-1:111122223333:table/checkingAccounts",
        "arn:aws:dynamodb:us-east-1:111122223333:table/checkingAccounts/
*",
        "arn:aws:dynamodb:us-east-1:111122223333:table/
transactionHistory",
        "arn:aws:dynamodb:us-east-1:111122223333:table/
transactionHistory/*"
      ]
    }
  ]
}
```

Transazioni

Per questo esempio, il contesto è una classica transazione bancaria, in cui useremo `TransactWriteItems` per:

- Trasferire denaro dai conti di deposito ai conti correnti
- Generare nuovi record di transazione per ogni transazione

Quindi, useremo `TransactGetItems` per recuperare i dettagli dai conti di deposito ai conti correnti.

Warning

`TransactWriteItems` non è supportato se utilizzato per il rilevamento e la risoluzione dei conflitti. Queste impostazioni devono essere disabilitate per evitare possibili errori.

Definiamo il nostro schema GraphQL come segue:

```
type SavingAccount {
  accountNumber: String!
  username: String
  balance: Float
}

type CheckingAccount {
  accountNumber: String!
  username: String
  balance: Float
}

type TransactionHistory {
  transactionId: ID!
  from: String
  to: String
  amount: Float
}

type TransactionResult {
  savingAccounts: [SavingAccount]
  checkingAccounts: [CheckingAccount]
  transactionHistory: [TransactionHistory]
```

```
}

input SavingAccountInput {
  accountNumber: String!
  username: String
  balance: Float
}

input CheckingAccountInput {
  accountNumber: String!
  username: String
  balance: Float
}

input TransactionInput {
  savingAccountNumber: String!
  checkingAccountNumber: String!
  amount: Float!
}

type Query {
  getAccounts(savingAccountNumbers: [String], checkingAccountNumbers: [String]):
  TransactionResult
}

type Mutation {
  populateAccounts(savingAccounts: [SavingAccountInput], checkingAccounts:
  [CheckingAccountInput]): TransactionResult
  transferMoney(transactions: [TransactionInput]): TransactionResult
}
```

TransactWriteItems - Compila gli account

Al fine di trasferire denaro tra gli account, abbiamo bisogno di popolare la tabella con i dettagli. Per farlo, useremo l'operazione GraphQL `Mutation.populateAccounts`.

Nella sezione Schema, fai clic su **Allega** accanto all'`Mutation.populateAccounts` operazione. Scegli l'origine `TransactTutorial` dati e scegli **Crea**.

Ora usa il seguente codice:

```
import { util } from '@aws-appsync/utils'
```

```

export function request(ctx) {
  const { savingAccounts, checkingAccounts } = ctx.args

  const savings = savingAccounts.map(({ accountNumber, ...rest }) => {
    return {
      table: 'savingAccounts',
      operation: 'PutItem',
      key: util.dynamodb.toMapValues({ accountNumber }),
      attributeValues: util.dynamodb.toMapValues(rest),
    }
  })

  const checkings = checkingAccounts.map(({ accountNumber, ...rest }) => {
    return {
      table: 'checkingAccounts',
      operation: 'PutItem',
      key: util.dynamodb.toMapValues({ accountNumber }),
      attributeValues: util.dynamodb.toMapValues(rest),
    }
  })

  return {
    version: '2018-05-29',
    operation: 'TransactWriteItems',
    transactItems: [...savings, ...checkings],
  }
}

export function response(ctx) {
  if (ctx.error) {
    util.error(ctx.error.message, ctx.error.type, null, ctx.result.cancellationReasons)
  }
  const { savingAccounts: sInput, checkingAccounts: cInput } = ctx.args
  const keys = ctx.result.keys
  const savingAccounts = sInput.map((_, i) => keys[i])
  const sLength = sInput.length
  const checkingAccounts = cInput.map((_, i) => keys[sLength + i])
  return { savingAccounts, checkingAccounts }
}

```

Salva il resolver e vai alla sezione Query della AWS AppSync console per popolare gli account.

Eseguire la mutazione seguente:

```
mutation populateAccounts {
```

```

populateAccounts (
  savingAccounts: [
    {accountNumber: "1", username: "Tom", balance: 100},
    {accountNumber: "2", username: "Amy", balance: 90},
    {accountNumber: "3", username: "Lily", balance: 80},
  ]
  checkingAccounts: [
    {accountNumber: "1", username: "Tom", balance: 70},
    {accountNumber: "2", username: "Amy", balance: 60},
    {accountNumber: "3", username: "Lily", balance: 50},
  ]) {
  savingAccounts {
    accountNumber
  }
  checkingAccounts {
    accountNumber
  }
}
}

```

Abbiamo popolato tre conti di risparmio e tre conti correnti in un'unica mutazione.

Utilizza la console DynamoDB per verificare che i dati vengano visualizzati nelle tabelle SavingAccounts e CheckingAccounts.

TransactWriteItems - Trasferisci denaro

Collega un resolver alla transferMoney mutazione con il seguente codice. Per ogni trasferimento, abbiamo bisogno di un modificatore di successo sia per il conto corrente che per il conto di risparmio e dobbiamo tenere traccia del trasferimento nelle transazioni.

```

import { util } from '@aws-appsync/utils'

export function request(ctx) {
  const transactions = ctx.args.transactions

  const savings = []
  const checkings = []
  const history = []
  transactions.forEach((t) => {
    const { savingAccountNumber, checkingAccountNumber, amount } = t
    savings.push({
      table: 'savingAccounts',

```

```

    operation: 'UpdateItem',
    key: util.dynamodb.toMapValues({ accountNumber: savingAccountNumber }),
    update: {
      expression: 'SET balance = balance - :amount',
      expressionValues: util.dynamodb.toMapValues({ ':amount': amount }),
    },
  })
  checkings.push({
    table: 'checkingAccounts',
    operation: 'UpdateItem',
    key: util.dynamodb.toMapValues({ accountNumber: checkingAccountNumber }),
    update: {
      expression: 'SET balance = balance + :amount',
      expressionValues: util.dynamodb.toMapValues({ ':amount': amount }),
    },
  })
  history.push({
    table: 'transactionHistory',
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({ transactionId: util.autoId() }),
    attributeValues: util.dynamodb.toMapValues({
      from: savingAccountNumber,
      to: checkingAccountNumber,
      amount,
    }),
  })
})

return {
  version: '2018-05-29',
  operation: 'TransactWriteItems',
  transactItems: [...savings, ...checkings, ...history],
}
}

export function response(ctx) {
  if (ctx.error) {
    util.error(ctx.error.message, ctx.error.type, null, ctx.result.cancellationReasons)
  }
  const tInput = ctx.args.transactions
  const tLength = tInput.length
  const keys = ctx.result.keys
  const savingAccounts = tInput.map((_, i) => keys[tLength * 0 + i])
  const checkingAccounts = tInput.map((_, i) => keys[tLength * 1 + i])

```

```
const transactionHistory = tInput.map((_, i) => keys[tLength * 2 + i])
return { savingAccounts, checkingAccounts, transactionHistory }
}
```

Ora, vai alla sezione Queries della AWS AppSync console ed esegui la mutazione TransferMoney come segue:

```
mutation write {
  transferMoney(
    transactions: [
      {savingAccountNumber: "1", checkingAccountNumber: "1", amount: 7.5},
      {savingAccountNumber: "2", checkingAccountNumber: "2", amount: 6.0},
      {savingAccountNumber: "3", checkingAccountNumber: "3", amount: 3.3}
    ]) {
    savingAccounts {
      accountNumber
    }
    checkingAccounts {
      accountNumber
    }
    transactionHistory {
      transactionId
    }
  }
}
```

Abbiamo inviato tre transazioni bancarie in un'unica mutazione. Utilizza la console DynamoDB per verificare che i dati vengano visualizzati nelle tabelle SavingAccounts, CheckingAccounts e TransactionHistory.

TransactGetItems - Recupera gli account

Per recuperare i dettagli dai conti di risparmio e correnti in un'unica richiesta transazionale, collegheremo un resolver all'operazione Query.getAccounts GraphQL sul nostro schema. Seleziona Allega, scegli la stessa fonte di TransactTutorial1 dati creata all'inizio del tutorial. Eseguire il seguente codice:

```
import { util } from '@aws-appsync/utils'

export function request(ctx) {
  const { savingAccountNumbers, checkingAccountNumbers } = ctx.args
```



```

const savings = savingAccountNumbers.map((accountNumber) => {
  return { table: 'savingAccounts', key: util.dynamodb.toMapValues({ accountNumber }) }
})
const checkings = checkingAccountNumbers.map((accountNumber) => {
  return { table: 'checkingAccounts', key:
util.dynamodb.toMapValues({ accountNumber }) }
})
return {
  version: '2018-05-29',
  operation: 'TransactGetItems',
  transactItems: [...savings, ...checkings],
}
}

export function response(ctx) {
  if (ctx.error) {
    util.error(ctx.error.message, ctx.error.type, null, ctx.result.cancellationReasons)
  }

  const { savingAccountNumbers: sInput, checkingAccountNumbers: cInput } = ctx.args
  const items = ctx.result.items
  const savingAccounts = sInput.map((_, i) => items[i])
  const sLength = sInput.length
  const checkingAccounts = cInput.map((_, i) => items[sLength + i])
  return { savingAccounts, checkingAccounts }
}

```

Salvare il resolver e passare alle sezioni Query della console AWS AppSync . Per recuperare i conti di risparmio e i conti correnti, esegui la seguente query:

```

query getAccounts {
  getAccounts(
    savingAccountNumbers: ["1", "2", "3"],
    checkingAccountNumbers: ["1", "2"]
  ) {
    savingAccounts {
      accountNumber
      username
      balance
    }
    checkingAccounts {
      accountNumber

```

```
    username
    balance
  }
}
```

Abbiamo dimostrato con successo l'uso delle transazioni DynamoDB utilizzando AWS AppSync

Utilizzo delle operazioni batch di DynamoDB in AWS AppSync

AWS AppSync supporta l'utilizzo di operazioni batch di Amazon DynamoDB su una o più tabelle in una singola regione. Le operazioni supportate sono `BatchGetItem`, `BatchPutItem` e `BatchDeleteItem`. Utilizzando queste funzionalità in AWS AppSync, puoi eseguire attività come:

- Passare un elenco di chiavi in una singola query e restituire i risultati da una tabella
- Leggere i record da una o più tabelle in una singola query
- Scrittura di record in blocco su una o più tabelle
- Scrittura o eliminazione condizionale di record in più tabelle che potrebbero avere una relazione

Le operazioni batch AWS AppSync presentano due differenze fondamentali rispetto alle operazioni non in batch:

- Il ruolo dell'origine dati deve disporre delle autorizzazioni per tutte le tabelle a cui accederà il resolver.
- La specifica della tabella per un resolver fa parte dell'oggetto della richiesta.

Batch a tabella singola

Warning

`BatchPutItem` e `BatchDeleteItem` non sono supportati se utilizzati per il rilevamento e la risoluzione dei conflitti. Queste impostazioni devono essere disabilitate per evitare possibili errori.

Per iniziare, creiamo una nuova API GraphQL. Nella AWS AppSync console, scegli **Create API**, **GraphQL APIs** e **Design da zero**. Assegna un nome all'API **BatchTutorial API**, scegli **Avanti** e nel

passaggio Specificare le risorse GraphQL, scegli Crea risorse GraphQL in un secondo momento e fai clic su Avanti. Controlla i tuoi dati e crea l'API. Vai alla pagina Schema e incolla lo schema seguente, notando che per la query inseriremo un elenco di IDs:

```
type Post {
  id: ID!
  title: String
}

input PostInput {
  id: ID!
  title: String
}

type Query {
  batchGet(ids: [ID]): [Post]
}

type Mutation {
  batchAdd(posts: [PostInput]): [Post]
  batchDelete(ids: [ID]): [Post]
}
```

Salva lo schema e scegli Crea risorse nella parte superiore della pagina. Scegli Usa il tipo esistente e seleziona il Post tipo. Assegna un nome alla tua tabella `Posts`. Assicurati che la chiave primaria sia impostata su `id`, deseleziona Genera automaticamente GraphQL (fornirai il tuo codice) e seleziona Crea. Per iniziare, AWS AppSync crea una nuova tabella DynamoDB e un'origine dati connessa alla tabella con i ruoli appropriati. Tuttavia, ci sono ancora un paio di autorizzazioni da aggiungere al ruolo. Vai alla pagina Fonti di dati e scegli la nuova fonte di dati. In Seleziona un ruolo esistente, noterai che un ruolo è stato creato automaticamente per la tabella. Prendi nota del ruolo (dovrebbe assomigliare a qualcosa del genere `appsync-ds-ddb-aaabbbcccddd-Posts`) e poi vai alla console IAM (<https://console.aws.amazon.com/iam/>). Nella console IAM, scegli Ruoli, quindi scegli il tuo ruolo dalla tabella. Nel tuo ruolo, in Politiche di autorizzazione, fai clic sul pulsante + "" accanto alla politica (dovrebbe avere un nome simile al nome del ruolo). Scegli Modifica nella parte superiore del pannello comprimibile quando viene visualizzata la politica. Devi aggiungere autorizzazioni batch alla tua politica, in particolare `dynamodb:BatchGetItem` e `dynamodb:BatchWriteItem`. Assomiglierà a questo:

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:DeleteItem",
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:Query",
        "dynamodb:Scan",
        "dynamodb:UpdateItem",
        "dynamodb:BatchWriteItem",
        "dynamodb:BatchGetItem"
      ],
      "Resource": [
        "arn:aws:dynamodb:us-east-1:111122223333:table/locationReadings",
        "arn:aws:dynamodb:us-east-1:111122223333:table/locationReadings/*",
        "arn:aws:dynamodb:us-east-1:111122223333:table/temperatureReadings",
        "arn:aws:dynamodb:us-east-1:111122223333:table/temperatureReadings/*"
      ]
    }
  ]
}
```

Scegli Avanti, quindi Salva modifiche. La tua politica dovrebbe ora consentire l'elaborazione in batch.

Tornando alla AWS AppSync console, vai alla pagina Schema e seleziona Allega accanto al `Mutation.batchAdd` campo. Crea il tuo resolver usando la `Posts` tabella come fonte di dati. Nell'editor di codice, sostituisci i gestori con lo snippet riportato di seguito. Questo frammento prende automaticamente ogni elemento del input `PostInput` tipo GraphQL e crea una mappa, necessaria per l'operazione: `BatchPutItem`

```
import { util } from "@aws-appsync/utils";

export function request(ctx) {
```

```

return {
  operation: "BatchPutItem",
  tables: {
    Posts: ctx.args.posts.map((post) => util.dynamodb.toMapValues(post)),
  },
};
}

export function response(ctx) {
  if (ctx.error) {
    util.error(ctx.error.message, ctx.error.type);
  }
  return ctx.result.data.Posts;
}

```

Vai alla pagina Query della AWS AppSync console ed esegui la seguente mutazione: `batchAdd`

```

mutation add {
  batchAdd(posts:[{
    id: 1 title: "Running in the Park"},{
    id: 2 title: "Playing fetch"
  }]){
    id
    title
  }
}

```

Dovresti vedere i risultati stampati sullo schermo; questo può essere convalidato esaminando la console DynamoDB per cercare i valori scritti nella tabella. `Posts`

Quindi, ripeti il processo di collegamento di un resolver ma per il `Query.batchGet` campo utilizzando la `Posts` tabella come fonte di dati. Sostituisci i gestori con il codice seguente. Questo prende automaticamente ogni voce nel tipo `ids: []` di GraphQL e crea una mappa, che risulta necessaria per l'operazione `BatchGetItem`:

```

import { util } from "@aws-appsync/utils";

export function request(ctx) {
  return {
    operation: "BatchGetItem",
    tables: {
      Posts: {

```

```

        keys: ctx.args.ids.map((id) => util.dynamodb.toMapValues({ id })),
        consistentRead: true,
      },
    },
  };
}

export function response(ctx) {
  if (ctx.error) {
    util.error(ctx.error.message, ctx.error.type);
  }
  return ctx.result.data.Posts;
}

```

Ora torna alla pagina Query della AWS AppSync console ed esegui la seguente query: `batchGet`

```

query get {
  batchGet(ids:[1,2,3]){
    id
    title
  }
}

```

Questo dovrebbe restituire i risultati per i due valori `id` aggiunti in precedenza. Nota che è stato restituito un `null` valore per il `id` con un valore di 3. Questo perché non c'era ancora nessun record nella `Posts` tabella con quel valore. Tieni inoltre presente che AWS AppSync restituisce i risultati nello stesso ordine delle chiavi passate alla query, che è una funzionalità aggiuntiva che viene AWS AppSync eseguita per conto dell'utente. Quindi, se passi `batchGet(ids:[1,3,2])`, vedrai che l'ordine è cambiato. È inoltre possibile sapere quale `id` ha restituito un valore `null`.

Infine, collega un altro resolver al `Mutation.batchDelete` campo usando la `Posts` tabella come fonte di dati. Sostituisci i gestori con il codice seguente. Questo prende automaticamente ogni voce nel tipo `ids: []` di GraphQL e crea una mappa, che risulta necessaria per l'operazione `BatchGetItem`:

```

import { util } from "@aws-appsync/utils";

export function request(ctx) {
  return {
    operation: "BatchDeleteItem",
    tables: {

```

```
    Posts: ctx.args.ids.map((id) => util.dynamodb.toMapValues({ id })),
  },
};
}

export function response(ctx) {
  if (ctx.error) {
    util.error(ctx.error.message, ctx.error.type);
  }
  return ctx.result.data.Posts;
}
```

Ora torna alla pagina Queries della AWS AppSync console ed esegui la seguente mutazione: `batchDelete`

```
mutation delete {
  batchDelete(ids:[1,2]){ id }
}
```

I record con id 1 e 2 dovrebbero essere eliminati. Se si esegue nuovamente la query `batchGet()` da una versione precedente, questi devono restituire `null`.

Batch multitavola

Warning

`BatchPutItem` e `BatchDeleteItem` non sono supportati se utilizzati per il rilevamento e la risoluzione dei conflitti. Queste impostazioni devono essere disabilitate per evitare possibili errori.

AWS AppSync consente inoltre di eseguire operazioni in batch su più tabelle. L'applicazione seguente è più complessa da costruire. Immagina di costruire un'app per la salute degli animali domestici in cui i sensori segnalano la posizione e la temperatura corporea dell'animale. I sensori sono dotati di batteria e tentano di connettersi alla rete a distanza di pochi minuti. Quando un sensore stabilisce una connessione, invia le sue letture alla nostra API. AWS AppSync li trigger quindi analizzano i dati in modo da presentare un pannello di controllo al proprietario dell'animale domestico, focalizzando l'attenzione sulla rappresentazione delle interazioni tra il sensore e l'archivio di dati di back-end.

Nella AWS AppSync console, scegli Create API, GraphQL APIs e Design da zero. Assegna un nome all'API **MultiBatchTutorial API**, scegli Avanti e nel passaggio Specificare le risorse GraphQL, scegli Crea risorse GraphQL in un secondo momento e fai clic su Avanti. Controlla i tuoi dati e crea l'API. Vai alla pagina Schema e incolla e salva lo schema seguente:

```

type Mutation {
  # Register a batch of readings
  recordReadings(tempReadings: [TemperatureReadingInput], locReadings:
[LocationReadingInput]): RecordResult
  # Delete a batch of readings
  deleteReadings(tempReadings: [TemperatureReadingInput], locReadings:
[LocationReadingInput]): RecordResult
}

type Query {
  # Retrieve all possible readings recorded by a sensor at a specific time
  getReadings(sensorId: ID!, timestamp: String!): [SensorReading]
}

type RecordResult {
  temperatureReadings: [TemperatureReading]
  locationReadings: [LocationReading]
}

interface SensorReading {
  sensorId: ID!
  timestamp: String!
}

# Sensor reading representing the sensor temperature (in Fahrenheit)
type TemperatureReading implements SensorReading {
  sensorId: ID!
  timestamp: String!
  value: Float
}

# Sensor reading representing the sensor location (lat,long)
type LocationReading implements SensorReading {
  sensorId: ID!
  timestamp: String!
  lat: Float
  long: Float
}

```



```
input TemperatureReadingInput {
  sensorId: ID!
  timestamp: String
  value: Float
}

input LocationReadingInput {
  sensorId: ID!
  timestamp: String
  lat: Float
  long: Float
}
```

Dobbiamo creare due tabelle DynamoDB:

- `locationReadings` memorizzerà le letture sulla posizione del sensore.
- `temperatureReadings` memorizzerà le letture della temperatura del sensore.

Entrambe le tabelle condivideranno la stessa struttura di chiave primaria: `sensorId` (`String`) come chiave di partizione e `timestamp` (`String`) come chiave di ordinamento.

Scegli Crea risorse nella parte superiore della pagina. Scegli Usa il tipo esistente e seleziona il `locationReadings` tipo. Assegna un nome alla tua tabella `locationReadings`. Assicurati che la chiave primaria sia impostata su `sensorId` e la chiave di ordinamento su `timestamp`. Deseleziona Genera automaticamente GraphQL (dovrai fornire il tuo codice) e seleziona Crea. Ripeti questo processo per `temperatureReadings` utilizzare `temperatureReadings` come nome del tipo e della tabella. Utilizzate gli stessi tasti di cui sopra.

Le nuove tabelle conterranno ruoli generati automaticamente. Ci sono ancora un paio di autorizzazioni da aggiungere a questi ruoli. Vai alla pagina Fonti di dati e scegli `locationReadings`. In Seleziona un ruolo esistente, puoi vedere il ruolo. Prendi nota del ruolo (dovrebbe assomigliare a qualcosa del genere `appsync-ds-ddb-aaabbbccdd-locationReadings`) e poi vai alla console IAM (<https://console.aws.amazon.com/iam/>). Nella console IAM, scegli Ruoli, quindi scegli il tuo ruolo dalla tabella. Nel tuo ruolo, in Politiche di autorizzazione, fai clic sul pulsante + "" accanto alla politica (dovrebbe avere un nome simile al nome del ruolo). Scegli Modifica nella parte superiore del pannello comprimibile quando viene visualizzata la politica. È necessario aggiungere autorizzazioni a questa politica. Avrà un aspetto simile a questo:

Scegli Avanti, quindi Salva modifiche. Ripeti questo processo per l'origine `temperatureReadings` dati utilizzando lo stesso frammento di policy riportato sopra.

BatchPutItem - Registrazione delle letture del sensore

I sensori devono essere in grado di inviare le loro letture una volta stabilita la connessione a Internet. Il campo `Mutation.recordReadings` di GraphQL è l'API che si userà per farlo. Dovremo aggiungere un resolver a questo campo.

Nella pagina Schema della AWS AppSync console, seleziona Allega accanto al `Mutation.recordReadings` campo. Nella schermata successiva, crea il tuo resolver usando la `locationReadings` tabella come fonte di dati.

Dopo aver creato il resolver, sostituisci i gestori con il seguente codice nell'editor. Questa `BatchPutItem` operazione ci consente di specificare più tabelle:

```
import { util } from '@aws-appsync/utils'

export function request(ctx) {
  const { locReadings, tempReadings } = ctx.args
  const locationReadings = locReadings.map((loc) => util.dynamodb.toMapValues(loc))
  const temperatureReadings = tempReadings.map((tmp) => util.dynamodb.toMapValues(tmp))

  return {
    operation: 'BatchPutItem',
    tables: {
      locationReadings,
      temperatureReadings,
    },
  }
}

export function response(ctx) {
  if (ctx.error) {
    util.appendError(ctx.error.message, ctx.error.type)
  }
  return ctx.result.data
}
```

Con le operazioni in batch, possono essere presenti entrambi gli errori e i risultati restituiti dalla chiamata. In questo caso, è possibile eseguire liberamente ulteriori operazioni di gestione degli errori.

Note

L'uso di `utils.appendError()` è simile a `util.error()`, con la principale distinzione che non interrompe la valutazione del gestore della richiesta o della risposta. Segnala invece che c'è stato un errore nel campo ma consente di valutare il gestore e di conseguenza di restituire i dati al chiamante. Si consiglia di utilizzarlo `utils.appendError()` quando l'applicazione deve restituire risultati parziali.

Salva il resolver e vai alla pagina Query nella console. AWS AppSync Ora possiamo inviare alcune letture dei sensori.

Eseguire la mutazione seguente:

```
mutation sendReadings {
  recordReadings(
    tempReadings: [
      {sensorId: 1, value: 85.5, timestamp: "2018-02-01T17:21:05.000+08:00"},
      {sensorId: 1, value: 85.7, timestamp: "2018-02-01T17:21:06.000+08:00"},
      {sensorId: 1, value: 85.8, timestamp: "2018-02-01T17:21:07.000+08:00"},
      {sensorId: 1, value: 84.2, timestamp: "2018-02-01T17:21:08.000+08:00"},
      {sensorId: 1, value: 81.5, timestamp: "2018-02-01T17:21:09.000+08:00"}
    ]
    locReadings: [
      {sensorId: 1, lat: 47.615063, long: -122.333551, timestamp:
"2018-02-01T17:21:05.000+08:00"},
      {sensorId: 1, lat: 47.615163, long: -122.333552, timestamp:
"2018-02-01T17:21:06.000+08:00"},
      {sensorId: 1, lat: 47.615263, long: -122.333553, timestamp:
"2018-02-01T17:21:07.000+08:00"},
      {sensorId: 1, lat: 47.615363, long: -122.333554, timestamp:
"2018-02-01T17:21:08.000+08:00"},
      {sensorId: 1, lat: 47.615463, long: -122.333555, timestamp:
"2018-02-01T17:21:09.000+08:00"}
    ]) {
    locationReadings {
      sensorId
      timestamp
      lat
      long
    }
    temperatureReadings {
```

```
    sensorId
    timestamp
    value
  }
}
```

Abbiamo inviato dieci letture dei sensori in un'unica mutazione con letture suddivise su due tabelle. Utilizza la console DynamoDB per verificare che i dati vengano visualizzati sia nelle tabelle che nelle tabelle. `locationReadings` `temperatureReadings`

BatchDeleteItem - Eliminazione delle letture dei sensori

Allo stesso modo, avremmo anche bisogno di poter eliminare lotti di letture dei sensori. Utilizzare il campo `Mutation.deleteReadings` di GraphQL per questo scopo. Nella pagina Schema della AWS AppSync console, seleziona `Allega` accanto al `Mutation.deleteReadings` campo. Nella schermata successiva, crea il tuo resolver usando la `locationReadings` tabella come fonte di dati.

Dopo aver creato il resolver, sostituisci i gestori nell'editor di codice con lo snippet riportato di seguito. In questo resolver, utilizziamo un mappatore di funzioni di supporto che estrae il e il dagli input forniti.

`sensorId` `timestamp`

```
import { util } from '@aws-appsync/utils'

export function request(ctx) {
  const { locReadings, tempReadings } = ctx.args
  const mapper = ({ sensorId, timestamp }) => util.dynamodb.toMapValues({ sensorId,
    timestamp })

  return {
    operation: 'BatchDeleteItem',
    tables: {
      locationReadings: locReadings.map(mapper),
      temperatureReadings: tempReadings.map(mapper),
    },
  }
}

export function response(ctx) {
  if (ctx.error) {
    util.appendError(ctx.error.message, ctx.error.type)
  }
}
```

```
    return ctx.result.data
  }
```

Salva il resolver e vai alla pagina Query nella console. AWS AppSync Ora, eliminiamo un paio di letture del sensore.

Eseguire la mutazione seguente:

```
mutation deleteReadings {
  # Let's delete the first two readings we recorded
  deleteReadings(
    tempReadings: [{sensorId: 1, timestamp: "2018-02-01T17:21:05.000+08:00"}]
    locReadings: [{sensorId: 1, timestamp: "2018-02-01T17:21:05.000+08:00"}]) {
    locationReadings {
      sensorId
      timestamp
      lat
      long
    }
    temperatureReadings {
      sensorId
      timestamp
      value
    }
  }
}
```

Note

A differenza dell'operazione `DeleteItem`, nella risposta non viene restituita la voce completamente eliminata. Viene restituita solo la chiave passata. Per ulteriori informazioni, consulta il [riferimento alla funzione `BatchDeleteItem` in JavaScript resolver per DynamoDB](#).

Verifica tramite la console DynamoDB che queste due letture siano state eliminate dalle tabelle `locationReadings` e `temperatureReadings`.

BatchGetItem - Recupera le letture

Un'altra operazione comune per la nostra app sarebbe quella di recuperare le letture di un sensore in un momento specifico. Allegare un resolver al campo `Query.getReadings` di GraphQL.

sullo schema. Nella pagina Schema della AWS AppSync console, seleziona **Allega accanto** al `Query.getReadings` campo. Nella schermata successiva, crea il tuo resolver usando la `locationReadings` tabella come fonte di dati.

Usiamo il codice seguente:

```
import { util } from '@aws-appsync/utils'

export function request(ctx) {
  const keys = [util.dynamodb.toMapValues(ctx.args)]
  const consistentRead = true
  return {
    operation: 'BatchGetItem',
    tables: {
      locationReadings: { keys, consistentRead },
      temperatureReadings: { keys, consistentRead },
    },
  }
}

export function response(ctx) {
  if (ctx.error) {
    util.appendError(ctx.error.message, ctx.error.type)
  }
  const { locationReadings: locs, temperatureReadings: temps } = ctx.result.data

  return [
    ...locs.map((l) => ({ ...l, __typename: 'LocationReading' })),
    ...temps.map((t) => ({ ...t, __typename: 'TemperatureReading' })),
  ]
}
```

Salva il resolver e vai alla pagina Query nella console. AWS AppSync Ora, recuperiamo le letture dei nostri sensori.

Eseguire la query seguente:

```
query getReadingsForSensorAndTime {
  # Let's retrieve the very first two readings
  getReadings(sensorId: 1, timestamp: "2018-02-01T17:21:06.000+08:00") {
    sensorId
    timestamp
  }
}
```

```
...on TemperatureReading {  
  value  
}  
...on LocationReading {  
  lat  
  long  
}  
}  
}
```

Abbiamo dimostrato con successo l'uso delle operazioni batch di DynamoDB utilizzando AWS AppSync

Gestione degli errori

Nel AWS AppSync, le operazioni sulle sorgenti dati a volte possono restituire risultati parziali. Risultati parziali è il termine che si userà per indicare quando l'output di un'operazione comprende alcuni dati e un errore. Poiché la gestione degli errori è intrinsecamente specifica dell'applicazione, AWS AppSync offre l'opportunità di gestire gli errori nel gestore delle risposte. L'errore di chiamata del resolver, se presente, è disponibile nel contesto come `ctx.error`. Gli errori di chiamata comprendono sempre un messaggio e un tipo, accessibili come proprietà `ctx.error.message` e `ctx.error.type`. Nel gestore delle risposte, è possibile gestire i risultati parziali in tre modi:

1. Ingoia l'errore di invocazione semplicemente restituendo i dati.
2. Genera un errore (`usingutil.error(...)`) interrompendo la valutazione del gestore, che non restituirà alcun dato.
3. Aggiungi un errore (`usandoutil.appendError(...)`) e restituisci anche i dati.

Dimostriamo ciascuno dei tre punti precedenti con le operazioni batch di DynamoDB.

Operazioni di batch di DynamoDB

Con le operazioni di batch di DynamoDB, è possibile che un batch sia completato parzialmente. Ovvero, è possibile che alcune delle chiavi o voci richieste non vengano elaborate. Se non AWS AppSync è in grado di completare un batch, nel contesto verranno impostati gli elementi non elaborati e un errore di invocazione.

La gestione degli errori verrà realizzata tramite la configurazione del campo `Query.getReadings` dall'operazione `BatchGetItem` dalla sezione precedente di questo tutorial. In questo momento,

È opportuno supporre che durante l'esecuzione del campo `Query.getReadings`, la tabella `temperatureReadings` DynamoDB esegua un throughput assegnato. DynamoDB ha generato `ProvisionedThroughputExceededException` un durante il secondo tentativo AWS AppSync per elaborare gli elementi rimanenti del batch.

Il seguente codice JSON rappresenta il contesto serializzato dopo la chiamata in batch di DynamoDB ma prima della chiamata del gestore delle risposte:

```
{
  "arguments": {
    "sensorId": "1",
    "timestamp": "2018-02-01T17:21:05.000+08:00"
  },
  "source": null,
  "result": {
    "data": {
      "temperatureReadings": [
        null
      ],
      "locationReadings": [
        {
          "lat": 47.615063,
          "long": -122.333551,
          "sensorId": "1",
          "timestamp": "2018-02-01T17:21:05.000+08:00"
        }
      ]
    },
    "unprocessedKeys": {
      "temperatureReadings": [
        {
          "sensorId": "1",
          "timestamp": "2018-02-01T17:21:05.000+08:00"
        }
      ],
      "locationReadings": []
    }
  },
  "error": {
    "type": "DynamoDB:ProvisionedThroughputExceededException",
    "message": "You exceeded your maximum allowed provisioned throughput for a table or for one or more global secondary indexes. (...)"
  },
}
```



```
"outErrors": []  
}
```

Alcune cose da tenere presente in questo contesto:

- L'errore di invocazione è stato impostato nel contesto in `ctx.error` by AWS AppSync e il tipo di errore è stato impostato su `DynamoDB:ProvisionedThroughputExceededException`
- I risultati vengono mappati per tabella sotto `ctx.result.data` anche se è presente un errore.
- Le chiavi che non sono state elaborate sono disponibili all'indirizzo.
`ctx.result.data.unprocessedKeys` In questo caso, non AWS AppSync è stato possibile recuperare l'elemento con la chiave (`SensorID:1, timestamp:2018-02-01T 17:21:05.000 + 08:00`) a causa dell'insufficiente velocità di trasmissione della tabella.

Note

Per `BatchPutItem`, è `ctx.result.data.unprocessedItems`. Per `BatchDeleteItem`, è `ctx.result.data.unprocessedKeys`.

È possibile gestire l'errore in tre modi diversi.

1. Assumere l'errore di chiamata

Restituire i dati senza gestire l'errore di chiamata assume efficacemente l'errore. In questo modo, il risultato per il campo di GraphQL è sempre positivo.

Il codice che scriviamo è familiare e si concentra solo sui dati dei risultati.

Gestore di risposte

```
export function response(ctx) {  
  return ctx.result.data  
}
```

Risposta GraphQL

```
{  
  "data": {  
    "getReadings": [  

```

```

    {
      "sensorId": "1",
      "timestamp": "2018-02-01T17:21:05.000+08:00",
      "lat": 47.615063,
      "long": -122.333551
    },
    {
      "sensorId": "1",
      "timestamp": "2018-02-01T17:21:05.000+08:00",
      "value": 85.5
    }
  ]
}

```

Non verrà aggiunto nessun errore alla risposta di errore poiché sono stati eseguiti solo i dati.

2. Generazione di un errore per interrompere l'esecuzione del gestore di risposte

Quando gli errori parziali devono essere trattati come errori completi dal punto di vista del client, è possibile interrompere l'esecuzione del gestore di risposte per impedire la restituzione dei dati. Il metodo di utilità `util.error(...)` raggiunge esattamente questo comportamento.

Codice del gestore di risposte

```

export function response(ctx) {
  if (ctx.error) {
    util.error(ctx.error.message, ctx.error.type, null,
      ctx.result.data.unprocessedKeys);
  }
  return ctx.result.data;
}

```

Risposta GraphQL

```

{
  "data": {
    "getReadings": null
  },
  "errors": [
    {
      "path": [
        "getReadings"
      ]
    }
  ]
}

```

```

    ],
    "data": null,
    "errorType": "DynamoDB:ProvisionedThroughputExceededException",
    "errorInfo": {
      "temperatureReadings": [
        {
          "sensorId": "1",
          "timestamp": "2018-02-01T17:21:05.000+08:00"
        }
      ],
      "locationReadings": []
    },
    "locations": [
      {
        "line": 58,
        "column": 3
      }
    ],
    "message": "You exceeded your maximum allowed provisioned throughput for a table
or for one or more global secondary indexes. (...)"
  }
]
}

```

Sebbene alcuni risultati possano essere stati restituiti dall'operazione in batch di DynamoDB, si è scelto di generare un errore in modo tale che il campo `getReadings` di GraphQL sia nullo e l'errore venga aggiunto al blocco di errori della risposta GraphQL.

3. Aggiunta di un errore per restituire sia i dati sia gli errori

In alcuni casi, per fornire una migliore esperienza utente, le applicazioni possono restituire risultati parziali e notificare ai client le voci non elaborate. I client possono decidere di implementare un nuovo tentativo oppure di rimandare l'errore all'utente finale. `util.appendError(...)` È il metodo di utilità che abilita questo comportamento consentendo al progettista dell'applicazione di aggiungere errori al contesto senza interferire con la valutazione del gestore della risposta. Dopo aver valutato il gestore della risposta, AWS AppSync elaborerà eventuali errori di contesto aggiungendoli al blocco `errors` della risposta GraphQL.

Codice del gestore di risposte

```

export function response(ctx) {
  if (ctx.error) {

```

```
    util.appendError(ctx.error.message, ctx.error.type, null,
ctx.result.data.unprocessedKeys);
  }
  return ctx.result.data;
}
```

Abbiamo inoltrato sia l'errore di invocazione che l'unprocessedKeyselemento all'interno del blocco errors della risposta GraphQL. Il getReadings campo restituisce anche dati parziali dalla locationReadings tabella, come puoi vedere nella risposta seguente.

Risposta GraphQL

```
{
  "data": {
    "getReadings": [
      null,
      {
        "sensorId": "1",
        "timestamp": "2018-02-01T17:21:05.000+08:00",
        "value": 85.5
      }
    ]
  },
  "errors": [
    {
      "path": [
        "getReadings"
      ],
      "data": null,
      "errorType": "DynamoDB:ProvisionedThroughputExceededException",
      "errorInfo": {
        "temperatureReadings": [
          {
            "sensorId": "1",
            "timestamp": "2018-02-01T17:21:05.000+08:00"
          }
        ],
        "locationReadings": []
      },
      "locations": [
        {
          "line": 58,
          "column": 3
        }
      ]
    }
  ]
}
```

```
    }  
  ],  
  "message": "You exceeded your maximum allowed provisioned throughput for a table  
or for one or more global secondary indexes. (...)"  
}  
]  
}
```

Utilizzo di resolver HTTP in AWS AppSync

AWS AppSync consente di utilizzare fonti di dati supportate (ad esempio Amazon DynamoDB AWS Lambda, Amazon Service o OpenSearch Amazon Aurora) per eseguire varie operazioni, oltre a qualsiasi endpoint HTTP arbitrario per risolvere i campi GraphQL. Quando gli endpoint HTTP sono disponibili, è possibile connettersi a questi utilizzando un'origine dati. Quindi, è possibile configurare un resolver nello schema per eseguire operazioni GraphQL, ad esempio query, mutazioni e sottoscrizioni. Questo tutorial fornirà una guida di alcuni esempi comuni.

In questo tutorial utilizzi un'API REST (creata utilizzando Amazon API Gateway e Lambda) con un endpoint GraphQL AWS AppSync .

Creazione di un'API REST

Puoi utilizzare il seguente AWS CloudFormation modello per configurare un endpoint REST adatto a questo tutorial:



Lo AWS CloudFormation stack esegue i seguenti passaggi:

1. Imposta una funzione Lambda che contiene la logica di business per il tuo microservizio.
2. Imposta un'API REST API Gateway con la seguente combinazione endpoint/method/content di tipi:

Percorso risorsa API	Metodo HTTP	Tipo di contenuto supportato
/v1/utenti	POST	application/json
/v1/utenti	GET	application/json

Percorso risorsa API	Metodo HTTP	Tipo di contenuto supportato
/v1/utenti/1	GET	application/json
/v1/utenti/1	PUT	application/json
/v1/utenti/1	DELETE	application/json

Creazione dell'API GraphQL

Per creare l'API GraphQL in: AWS AppSync

1. Apri la AWS AppSync console e scegli Crea API.
2. Scegli GraphQL APIs e poi scegli Design da zero. Scegli Next (Successivo).
3. Per il nome API, digita UserData. Scegli Next (Successivo).
4. Scegli Create GraphQL resources later. Scegli Next (Successivo).
5. Controlla i tuoi input e scegli Crea API.

La AWS AppSync console crea una nuova API GraphQL utilizzando la modalità di autenticazione della chiave API. Puoi utilizzare la console per configurare ulteriormente l'API GraphQL ed eseguire le richieste.

Creazione di uno schema GraphQL

Ora che hai un'API GraphQL, è possibile creare uno schema GraphQL. Nell'editor di schema della AWS AppSync console, usa lo snippet seguente:

```
type Mutation {  
  addUser(userInput: UserInput!): User  
  deleteUser(id: ID!): User  
}  
  
type Query {  
  getUser(id: ID): User  
  listUser: [User!]!  
}  
  
type User {
```

```
    id: ID!
    username: String!
    firstname: String
    lastname: String
    phone: String
    email: String
  }

  input UserInput {
    id: ID!
    username: String!
    firstname: String
    lastname: String
    phone: String
    email: String
  }
```

Configura la tua fonte di dati HTTP

Per configurare l'origine dati HTTP, effettua le seguenti operazioni:

1. Nella pagina Sorgenti dati dell'API AWS AppSync GraphQL, scegli Crea origine dati.
2. Inserisci un nome per la fonte di dati, ad esempio `HTTP_Example`.
3. In Tipo di origine dati, scegli Endpoint HTTP.
4. Imposta l'endpoint sull'endpoint API Gateway creato all'inizio del tutorial. Puoi trovare l'endpoint generato dallo stack accedendo alla console Lambda e trovando l'applicazione in Applicazioni. All'interno delle impostazioni dell'applicazione, dovresti vedere un endpoint API che fungerà da endpoint. AWS AppSync Assicurati di non includere il nome dello stage come parte dell'endpoint. Ad esempio, se il tuo endpoint lo fosse `https://aaabbbcccd.execute-api.us-east-1.amazonaws.com/v1`, dovresti digitare `https://aaabbbcccd.execute-api.us-east-1.amazonaws.com`.

Note

Al momento, solo gli endpoint pubblici sono supportati da AWS AppSync. Per ulteriori informazioni sulle autorità di certificazione riconosciute dal AWS AppSync servizio, consulta [Autorità di certificazione \(CA\) riconosciute da AWS AppSync per gli endpoint HTTPS](#).

Configurazione dei resolver

In questo passaggio, collegherai l'origine dati HTTP alle query `getUser` and `addUser`.

Per configurare il `getUser` resolver:

1. Nella tua API AWS AppSync GraphQL, scegli la scheda Schema.
2. A destra dell'editor dello schema, nel riquadro Resolvers e sotto il tipo di query, trova il **`getUser`** campo e scegli **Allega**.
3. Mantieni invariato il tipo di resolver e il runtime `inalteratoUnit`. `APPSYNC_JS`
4. Nel Nome dell'origine dati, scegli l'endpoint HTTP creato in precedenza.
5. Scegli **Create (Crea)**.
6. Nell'editor di codice Resolver, aggiungi il seguente frammento come gestore delle richieste:

```
import { util } from '@aws-appsync/utils'

export function request(ctx) {
  return {
    version: '2018-05-29',
    method: 'GET',
    params: {
      headers: {
        'Content-Type': 'application/json',
      },
    },
    resourcePath: `/v1/users/${ctx.args.id}`,
  }
}
```

7. Aggiungi il seguente frammento come gestore delle risposte:

```
export function response(ctx) {
  const { statusCode, body } = ctx.result
  // if response is 200, return the response
  if (statusCode === 200) {
    return JSON.parse(body)
  }
  // if response is not 200, append the response to error block.
  util.appendError(body, statusCode)
}
```


8. Scegliere la scheda Query quindi eseguire la seguente query:

```
query GetUser{
  getUser(id:1){
    id
    username
  }
}
```

Viene restituita la risposta seguente:

```
{
  "data": {
    "getUser": {
      "id": "1",
      "username": "nadia"
    }
  }
}
```

Per configurare il resolver: `addUser`

1. Scegli la scheda Schema.
2. A destra dell'editor dello schema, nel riquadro Resolver e sotto il tipo di query, trova il **addUser** campo e scegli Allega.
3. Mantieni invariato il tipo di resolver e il runtime `inalteratoUnit`. `APPSYNC_JS`
4. Nel Nome dell'origine dati, scegli l'endpoint HTTP creato in precedenza.
5. Scegli Create (Crea).
6. Nell'editor di codice Resolver, aggiungi il seguente frammento come gestore delle richieste:

```
export function request(ctx) {
  return {
    "version": "2018-05-29",
    "method": "POST",
    "resourcePath": "/v1/users",
    "params": {
      "headers": {
        "Content-Type": "application/json"
      }
    },
  },
}
```

```
        "body": ctx.args.userInput
      }
    }
  }
}
```

7. Aggiungi il seguente frammento come gestore delle risposte:

```
export function response(ctx) {
  if(ctx.error) {
    return util.error(ctx.error.message, ctx.error.type)
  }
  if (ctx.result.statusCode == 200) {
    return ctx.result.body
  } else {
    return util.appendError(ctx.result.body, "ctx.result.statusCode")
  }
}
```

8. Scegliere la scheda Query quindi eseguire la seguente query:

```
mutation addUser{
  addUser(userInput:{
    id:"2",
    username:"shaggy"
  }){
    id
    username
  }
}
```

Se esegui nuovamente la `getUser` query, dovrebbe restituire la seguente risposta:

```
{
  "data": {
    "getUser": {
      "id": "2",
      "username": "shaggy"
    }
  }
}
```

Richiamo dei servizi AWS

È possibile utilizzare i resolver HTTP per configurare un'interfaccia API GraphQL per i servizi AWS. Le richieste HTTP AWS devono essere firmate con il [processo Signature Version 4](#) in modo da AWS poter identificare chi le ha inviate. AWS AppSync calcola la firma per tuo conto quando associ un ruolo IAM all'origine dati HTTP.

Fornisci due componenti aggiuntivi per richiamare i AWS servizi con resolver HTTP:

- Un ruolo IAM con autorizzazioni per chiamare il servizio AWS APIs
- Configurazione della firma nell'origine dati

Ad esempio, se desideri chiamare l'[ListGraphqlApis operazione](#) con resolver HTTP, devi prima [creare un ruolo IAM](#) che AWS AppSync presuppone la seguente policy allegata:

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "appsync:ListGraphqlApis"
      ],
      "Effect": "Allow",
      "Resource": "*"
    }
  ]
}
```

Quindi, crea l'origine dati HTTP per AWS AppSync. In questo esempio, chiami AWS AppSync nella regione Stati Uniti occidentali (Oregon). Imposta la seguente configurazione HTTP in un file denominato `http.json`, che include la regione di firma e il nome del servizio:

```
{
  "endpoint": "https://appsync.us-west-2.amazonaws.com/",
  "authorizationConfig": {
    "authorizationType": "AWS_IAM",
```

```
    "awsIamConfig": {
      "signingRegion": "us-west-2",
      "signingServiceName": "appsync"
    }
  }
}
```

Quindi, utilizzate il AWS CLI per creare l'origine dati con un ruolo associato come segue:

```
aws appsync create-data-source --api-id <API-ID> \
                               --name AWSAppSync \
                               --type HTTP \
                               --http-config file:///http.json \
                               --service-role-arn <ROLE-ARN>
```

Quando colleghi un resolver al campo dello schema, usa il seguente modello di mappatura delle richieste per chiamare: AWS AppSync

```
{
  "version": "2018-05-29",
  "method": "GET",
  "resourcePath": "/v1/apis"
}
```

Quando esegui una query GraphQL per questa fonte di dati, AWS AppSync firma la richiesta utilizzando il ruolo che hai fornito e include la firma nella richiesta. La query restituisce un elenco di AWS AppSync GraphQL APIs nel tuo account in quella AWS regione.

Utilizzo di Aurora PostgreSQL con Data API in AWS AppSync

Scopri come connettere l'API GraphQL ai database Aurora PostgreSQL utilizzando. AWS AppSync Questa integrazione consente di creare applicazioni scalabili e basate sui dati eseguendo query e mutazioni SQL tramite operazioni GraphQL. AWS AppSync fornisce un'origine dati per l'esecuzione di istruzioni SQL su cluster Amazon Aurora abilitati con un'API Data. Puoi utilizzare AWS AppSync i resolver per eseguire istruzioni SQL sull'API dei dati con query, mutazioni e sottoscrizioni GraphQL.

Prima di iniziare questo tutorial, è necessario avere una conoscenza di base AWS dei servizi e dei concetti di GraphQL.

 Note

In questo tutorial viene utilizzata la regione US-EAST-1.

Argomenti

- [Configura il tuo database Aurora PostgreSQL](#)
- [Creazione del database e della tabella](#)
- [Creazione di uno schema GraphQL](#)
- [Resolver per RDS](#)
- [Eliminazione del cluster](#)

Configura il tuo database Aurora PostgreSQL

Prima di aggiungere un'origine dati Amazon RDS AWS AppSync, procedi come segue.

1. Abilita un'API dati su un cluster Aurora Serverless v2.
2. Configura un segreto utilizzando Gestione dei segreti AWS
3. Crea il cluster utilizzando il seguente AWS CLI comando.

```
aws rds create-db-cluster \  
    --db-cluster-identifier appsync-tutorial \  
    --engine aurora-postgresql \  
    --engine-version 16.6 \  
    --serverless-v2-scaling-configuration MinCapacity=0,MaxCapacity=1 \  
    --master-username USERNAME \  
    --master-user-password COMPLEX_PASSWORD \  
    --enable-http-endpoint
```

Verrà restituito un ARN per il cluster. Dopo aver creato un cluster, è necessario aggiungere un'istanza Serverless v2 con il comando seguente AWS CLI .

```
aws rds create-db-instance \  
    --db-cluster-identifier appsync-tutorial \  
    --db-instance-identifier appsync-tutorial-instance-1 \  
    --db-instance-class db.serverless \  
    --engine aurora-postgresql
```

```
--engine aurora-postgresql
```

Note

L'attivazione di questi endpoint richiede tempo. Puoi controllarne lo stato nella console RDS nella scheda Connettività e sicurezza del cluster.

Controlla lo stato del cluster con il seguente AWS CLI comando.

```
aws rds describe-db-clusters \  
  --db-cluster-identifier appsync-tutorial \  
  --query "DBClusters[0].Status"
```

Crea un segreto tramite la Gestione dei segreti AWS console o AWS CLI con un file di input come il seguente utilizzando i USERNAME e COMPLEX_PASSWORD del passaggio precedente:

```
{  
  "username": "USERNAME",  
  "password": "COMPLEX_PASSWORD"  
}
```

Passalo come parametro a AWS CLI:

```
aws secretsmanager create-secret \  
  --name appsync-tutorial-rds-secret \  
  --secret-string file://creds.json
```

Verrà restituito un ARN per il segreto. Prendi nota dell'ARN del tuo cluster Aurora Serverless v2 e di Secret per dopo quando crei un'origine dati nella console. AWS AppSync

Creazione del database e della tabella

Innanzitutto, crea un database denominato TESTDB. In PostgreSQL, un database è un contenitore che contiene tabelle e altri oggetti SQL. Verifica che il cluster Aurora Serverless v2 sia configurato correttamente prima di aggiungerlo all'API. AWS AppSync Innanzitutto, crea un database TESTDB con il parametro seguente. `--sql`

```
aws rds-data execute-statement \  
  --sql
```

```
--resource-arn "arn:aws:rds:us-east-1:111122223333 ISN:cluster:appsync-tutorial" \
--secret-arn "arn:aws:secretsmanager:us-east-1:111122223333 ISN:secret:appsync-
tutorial-rds-secret" \
--sql "create DATABASE \"testdb\"" \
--database "postgres"
```

Se viene eseguito senza errori, aggiungete due tabelle con il `create table` comando:

```
aws rds-data execute-statement \
--resource-arn "arn:aws:rds:us-east-1:111122223333 ISN:cluster:appsync-tutorial" \
--secret-arn "arn:aws:secretsmanager:us-east-1:111122223333 ISN:secret:appsync-
tutorial-rds-secret" \
--database "testdb" \
--sql 'create table public.todos (id serial constraint todos_pk primary key,
description text not null, due date not null, "createdAt" timestamp default now());'

aws rds-data execute-statement \
--resource-arn "arn:aws:rds:us-east-1:111122223333 ISN:cluster:appsync-tutorial" \
--secret-arn "arn:aws:secretsmanager:us-east-1:111122223333 ISN:secret:appsync-
tutorial-rds-secret" \
--database "testdb" \
--sql 'create table public.tasks (id serial constraint tasks_pk primary key,
description varchar, "todoId" integer not null constraint tasks_todos_id_fk references
public.todos);'
```

In caso di successo, aggiungi il cluster come fonte di dati nella tua API.

Creazione di uno schema GraphQL

Ora che la tua API dati Aurora Serverless v2 è in esecuzione con tabelle configurate, creeremo uno schema GraphQL. Puoi creare rapidamente la tua API importando le configurazioni delle tabelle da un database esistente utilizzando la procedura guidata di creazione dell'API.

Per iniziare:

1. Nella AWS AppSync console, scegli Crea API, quindi Inizia con un cluster Amazon Aurora.
2. Specificate i dettagli dell'API, come il nome dell'API, quindi selezionate il database per generare l'API.
3. Scegli il tuo database. Se necessario, aggiorna la regione, quindi scegli il cluster Aurora e il database TESTDB.
4. Scegli il tuo segreto, quindi scegli Importa.

5. Una volta scoperte le tabelle, aggiorna i nomi dei tipi. Passa Todos a Todo e Tasks a Task.
6. Visualizza l'anteprima dello schema generato scegliendo Anteprima schema. Il tuo schema sarà simile al seguente:

```

type Todo {
  id: Int!
  description: String!
  due: AWSDate!
  createdAt: String
}

type Task {
  id: Int!
  todoId: Int!
  description: String
}

```

7. Per il ruolo, puoi AWS AppSync creare un nuovo ruolo o crearne uno con una politica simile a quella seguente:

JSON

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "rds-data:ExecuteStatement"
      ],
      "Resource": [
        "arn:aws:rds:us-east-1:111122223333:cluster:appsync-tutorial",
        "arn:aws:rds:us-east-1:111122223333:cluster:appsync-tutorial:*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "secretsmanager:GetSecretValue"
      ],
      "Resource": [
        "arn:aws:secretsmanager:us-east-1:111122223333:secret:appsync-tutorial-rds-secret",

```



```

        "arn:aws:secretsmanager:us-east-1:111122223333:secret:appsync-
tutorial-rds-secret:*"
      ]
    }
  ]
}

```

Tieni presente che ci sono due dichiarazioni in questa politica a cui concedi l'accesso al ruolo. La prima risorsa è il cluster Aurora e la seconda è l' Gestione dei segreti AWS ARN.

Scegli Avanti, esamina i dettagli di configurazione, quindi scegli Crea API. Ora hai un'API completamente operativa. Puoi rivedere tutti i dettagli della tua API nella pagina Schema.

Resolver per RDS

Il flusso di creazione dell'API ha creato automaticamente i resolver per interagire con i nostri tipi. Se guardi la pagina Schema, troverai dei resolver, alcuni dei seguenti resolver.

- Crea `Mutation.createTodo` un tramite il campo. `todo`
- Aggiorna un todo messaggio tramite il `Mutation.updateTodo` campo.
- Elimina un todo tramite il `Mutation.deleteTodo` campo.
- Acquistane uno todo tramite il `Query.getTodo` campo.
- Elenca tutto todos tramite il `Query.listTodos` campo.

Troverai campi e resolver simili allegati per il tipo. Task Diamo un'occhiata più da vicino ad alcuni resolver.

Mutazione. CreateToDo

Dall'editor dello schema nella AWS AppSync console, sul lato destro, scegli accanto a. `testdb createTodo(...)`: Todo Il codice del resolver utilizza la `insert` funzione del `rds` modulo per creare dinamicamente un'istruzione `insert` che aggiunge dati alla tabella. `todos` Poiché stiamo lavorando con Postgres, possiamo sfruttare l'`returning` istruzione per recuperare i dati inseriti.

Aggiorna il seguente resolver per specificare correttamente il tipo di campo. DATE due

```
import { util } from '@aws-appsync/utils';
```

```
import { insert, createPgStatement, toJsonObject, typeHint } from '@aws-appsync/utils/
rds';

export function request(ctx) {
  const { input } = ctx.args;
  // if a due date is provided, cast is as `DATE`
  if (input.due) {
    input.due = typeHint.DATE(input.due)
  }
  const insertStatement = insert({
    table: 'todos',
    values: input,
    returning: '*',
  });
  return createPgStatement(insertStatement)
}

export function response(ctx) {
  const { error, result } = ctx;
  if (error) {
    return util.appendError(
      error.message,
      error.type,
      result
    )
  }
  return toJsonObject(result)[0][0]
}
```

Salva il resolver. Il suggerimento sul tipo contrassegna due correttamente il nostro oggetto di input come tipo. DATE Ciò consente al motore Postgres di interpretare correttamente il valore. Quindi, aggiorna lo schema per rimuovere il id CreateTodo dall'input. Poiché il nostro database Postgres può restituire l'ID generato, puoi fare affidamento su di esso per la creazione e la restituzione del risultato come una singola richiesta come segue.

```
input CreateTodoInput {
  due: AWSDate!
  createdAt: String
  description: String!
}
```

Apporta la modifica e aggiorna lo schema. Vai all'editor Queries per aggiungere un elemento al database come segue.

```
mutation CreateTodo {  
  createTodo(input: {description: "Hello World!", due: "2023-12-31"}) {  
    id  
    due  
    description  
    createdAt  
  }  
}
```

Ottieni il seguente risultato.

```
{  
  "data": {  
    "createTodo": {  
      "id": 1,  
      "due": "2023-12-31",  
      "description": "Hello World!",  
      "createdAt": "2023-11-14 20:47:11.875428"  
    }  
  }  
}
```

query.listTodos

Dall'editor dello schema nella console, sul lato destro, scegli accanto a `testdb listTodos(id: ID!)`: Todo Il gestore delle richieste utilizza la funzione di utilità `select` per creare una richiesta in modo dinamico in fase di esecuzione.

```
export function request(ctx) {  
  const { filter = {}, limit = 100, nextToken } = ctx.args;  
  const offset = nextToken ? +util.base64Decode(nextToken) : 0;  
  const statement = select({  
    table: 'todos',  
    columns: '*',  
    limit,  
    offset,  
    where: filter,  
  });  
  return createPgStatement(statement)
```

```
}

```

Vogliamo filtrare in todos base alla due data. Aggiorniamo il resolver per trasmettere due i valori a. DATE Aggiorna l'elenco delle importazioni e il gestore delle richieste come segue.

```
import { util } from '@aws-appsync/utils';
import * as rds from '@aws-appsync/utils/rds';

export function request(ctx) {
  const { filter: where = {}, limit = 100, nextToken } = ctx.args;
  const offset = nextToken ? +util.base64Decode(nextToken) : 0;

  // if `due` is used in a filter, CAST the values to DATE.
  if (where.due) {
    Object.entries(where.due).forEach(([k, v]) => {
      if (k === 'between') {
        where.due[k] = v.map((d) => rds.typeHint.DATE(d));
      } else {
        where.due[k] = rds.typeHint.DATE(v);
      }
    });
  }

  const statement = rds.select({
    table: 'todos',
    columns: '*',
    limit,
    offset,
    where,
  });
  return rds.createPgStatement(statement);
}

export function response(ctx) {
  const {
    args: { limit = 100, nextToken },
    error,
    result,
  } = ctx;
  if (error) {
    return util.appendError(error.message, error.type, result);
  }
  const offset = nextToken ? +util.base64Decode(nextToken) : 0;

```

```
const items = rds.toJsonObject(result)[0];
const endOfResults = items?.length < limit;
const token = endOfResults ? null : util.base64Encode(`${offset + limit}`);
return { items, nextToken: token };
}
```

Nell'editor Queries, procedi come segue.

```
query LIST {
  listTodos(limit: 10, filter: {due: {between: ["2021-01-01", "2025-01-02"]}}) {
    items {
      id
      due
      description
    }
  }
}
```

Mutation.UpdateToDo

Puoi anche un. update Todo Dall'editor di Queries, aggiorniamo il nostro primo Todo elemento di id1.

```
mutation UPDATE {
  updateTodo(input: {id: 1, description: "edits"}) {
    description
    due
    id
  }
}
```

Nota che devi specificare id l'elemento che stai aggiornando. Puoi anche specificare una condizione per aggiornare solo un elemento che soddisfa condizioni specifiche. Ad esempio, potremmo voler modificare l'elemento solo se la descrizione inizia con la edits seguente.

```
mutation UPDATE {
  updateTodo(input: {id: 1, description: "edits: make a change"}, condition:
  {description: {beginsWith: "edits"}}) {
    description
    due
    id
  }
}
```

```
}  
}
```

Proprio come abbiamo gestito le nostre `list` operazioni create e, allo stesso modo, possiamo aggiornare il nostro resolver per indirizzare il due campo `a.DATE`. Salva queste modifiche come segue. `updateTodo`

```
import { util } from '@aws-appsync/utils';  
import * as rds from '@aws-appsync/utils/rds';  
  
export function request(ctx) {  
  const { input: { id, ...values }, condition = {}, } = ctx.args;  
  const where = { ...condition, id: { eq: id } };  
  
  // if `due` is used in a condition, CAST the values to DATE.  
  if (condition.due) {  
    Object.entries(condition.due).forEach(([k, v]) => {  
      if (k === 'between') {  
        condition.due[k] = v.map((d) => rds.typeHint.DATE(d));  
      } else {  
        condition.due[k] = rds.typeHint.DATE(v);  
      }  
    });  
  }  
  
  // if a due date is provided, cast is as `DATE`  
  if (values.due) {  
    values.due = rds.typeHint.DATE(values.due);  
  }  
  
  const updateStatement = rds.update({  
    table: 'todos',  
    values,  
    where,  
    returning: '*',  
  });  
  return rds.createPgStatement(updateStatement);  
}  
  
export function response(ctx) {  
  const { error, result } = ctx;  
  if (error) {  
    return util.appendError(error.message, error.type, result);  
  }  
}
```

```
}  
return rds.toJsonObject(result)[0][0];  
}
```

Ora prova un aggiornamento con una condizione:

```
mutation UPDATE {  
  updateTodo(  
    input: {  
      id: 1, description: "edits: make a change", due: "2023-12-12"},  
    condition: {  
      description: {beginsWith: "edits"}, due: {ge: "2023-11-08"}})  
    {  
      description  
      due  
      id  
    }  
  }  
}
```

Mutazione. deleteToDo

Puoi farlo delete con la mutazione. Todo deleteTodo Funziona come la updateTodo mutazione ed è necessario specificare id l'elemento che si desidera eliminare come segue.

```
mutation DELETE {  
  deleteTodo(input: {id: 1}) {  
    description  
    due  
    id  
  }  
}
```

Scrittura di interrogazioni personalizzate

Abbiamo usato le utilità del `rds` modulo per creare le nostre istruzioni SQL. Possiamo anche scrivere la nostra dichiarazione statica personalizzata per interagire con il nostro database. Innanzitutto, aggiorna lo schema per rimuovere il `id` campo dall'`CreateTaskInput`.

```
input CreateTaskInput {  
  todoId: Int!  
  description: String
```

```
}

```

Quindi, crea un paio di attività. Un'attività ha una relazione a chiave esterna con Todo la seguente.

```
mutation TASKS {
  a: createTask(input: {todoId: 2, description: "my first sub task"}) { id }
  b:createTask(input: {todoId: 2, description: "another sub task"}) { id }
  c: createTask(input: {todoId: 2, description: "a final sub task"}) { id }
}
```

Crea un nuovo campo nel tuo Query tipo chiamato `getTodoAndTasks` come segue.

```
getTodoAndTasks(id: Int!): Todo
```

Aggiungi un `tasks` campo al `Todo` tipo come segue.

```
type Todo {
  due: AWSDate!
  id: Int!
  createdAt: String
  description: String!
  tasks:TaskConnection
}
```

Salvare lo schema. Dall'editor di schemi nella console, sul lato destro, scegli `Attach Resolver for`. `getTodosAndTasks(id: Int!): Todo` Scegli la tua fonte di dati Amazon RDS. Aggiorna il tuo resolver con il seguente codice.

```
import { sql, createPgStatement,toJsonObject } from '@aws-appsync/utils/rds';

export function request(ctx) {
  return createPgStatement(
    sql`SELECT * from todos where id = ${ctx.args.id}`,
    sql`SELECT * from tasks where "todoId" = ${ctx.args.id}`);
}

export function response(ctx) {
  const result = toJsonObject(ctx.result);
  const todo = result[0][0];
  if (!todo) {
    return null;
  }
}
```



```
    }  
    todo.tasks = { items: result[1] };  
    return todo;  
  }  
}
```

In questo codice, utilizziamo il modello di `sql` tag per scrivere un'istruzione SQL a cui possiamo passare in sicurezza un valore dinamico in fase di esecuzione. `createPgStatement` può richiedere fino a due richieste SQL alla volta. Lo usiamo per inviare una richiesta per la nostra `todo` e un'altra per la nostra `tasks`. Avresti potuto farlo con una `JOIN` dichiarazione o con qualsiasi altro metodo. L'idea è quella di poter scrivere la propria istruzione SQL per implementare la logica aziendale. Per utilizzare la query nell'editor Queries, procedi come segue.

```
query TodoAndTasks {  
  getTodosAndTasks(id: 2) {  
    id  
    due  
    description  
    tasks {  
      items {  
        id  
        description  
      }  
    }  
  }  
}
```

Eliminazione del cluster

Important

L'eliminazione di un cluster è permanente. Esamina attentamente il tuo progetto prima di eseguire questa azione.

Per eliminare il cluster:

```
$ aws rds delete-db-cluster \  
  --db-cluster-identifier appsync-tutorial \  
  --skip-final-snapshot
```

Tutorial sui resolver VTL per AWS AppSync

Note

Ora supportiamo principalmente il runtime APPSYNC_JS e la relativa documentazione. [Prendi in considerazione l'utilizzo del runtime APPSYNC_JS e delle relative guide qui.](#)

Le sorgenti di dati e i resolver vengono utilizzati per AWS AppSync tradurre le richieste GraphQL e recuperare informazioni dalle tue risorse. AWS AppSync supporta il provisioning automatico e le connessioni con determinati tipi di fonti di dati. AWS AppSync supporta anche Amazon DynamoDB, AWS Lambda, database relazionali (Amazon Aurora Serverless), OpenSearch Amazon Service ed endpoint HTTP come fonti di dati. Puoi utilizzare un'API GraphQL con le tue AWS risorse esistenti o creare sorgenti di dati e resolver partendo da zero. Le sezioni seguenti hanno lo scopo di chiarire alcuni dei casi d'uso più comuni di GraphQL sotto forma di tutorial.

AWS AppSync utilizza modelli di mappatura scritti in Apache Velocity Template Language (VTL) per i resolver. [Per ulteriori informazioni sull'utilizzo dei modelli di mappatura, consulta il riferimento ai modelli di mappatura Resolver.](#) Ulteriori informazioni sull'utilizzo di VTL sono disponibili nella guida alla programmazione dei modelli di mappatura [Resolver](#).

AWS AppSync supporta il provisioning automatico di tabelle DynamoDB da uno schema GraphQL come descritto in Provision from schema (opzionale) e Launch a sample schema. Puoi anche importare i dati da una tabella DynamoDB esistente che creerà lo schema e conetterà i resolver. Questo è descritto in Importazione da Amazon DynamoDB (opzionale).

Argomenti

- [Creazione di una semplice applicazione post utilizzando i resolver DynamoDB](#)
- [Utilizzo dei AWS Lambda resolver in AWS AppSync](#)
- [Utilizzo dei OpenSearch resolver Amazon Service in AWS AppSync](#)
- [Utilizzo di resolver locali in AWS AppSync](#)
- [Combinazione di resolver GraphQL in AWS AppSync](#)
- [Utilizzo delle operazioni batch di DynamoDB in AWS AppSync](#)
- [Esecuzione di transazioni DynamoDB in AWS AppSync](#)
- [Utilizzo di resolver HTTP in AWS AppSync](#)

- [Utilizzo di Aurora Serverless v2 con AWS AppSync](#)
- [Utilizzo di risolutori di pipeline in AWS AppSync](#)
- [Utilizzo delle operazioni Delta Sync su sorgenti dati con versione in AWS AppSync](#)

Creazione di una semplice applicazione post utilizzando i resolver DynamoDB

Note

Ora supportiamo principalmente il runtime APPSYNC_JS e la relativa documentazione. [Prendi in considerazione l'utilizzo del runtime APPSYNC_JS e delle relative guide qui.](#)

Questo tutorial mostra come importare tabelle Amazon DynamoDB personalizzate e collegarle AWS AppSync a un'API GraphQL.

Puoi consentire il AWS AppSync provisioning delle risorse DynamoDB per tuo conto. In alternativa, se preferisci, puoi connettere le tabelle esistenti a uno schema GraphQL creando un'origine dati e un resolver. In entrambi i casi, potrai leggere e scrivere nel database DynamoDB tramite istruzioni GraphQL e sottoscrivere dati in tempo reale.

Dovrai eseguire operazioni specifiche perché le istruzioni GraphQL possano essere tradotte in operazioni di DynamoDB e perché le risposte possano essere ritradotte in GraphQL. Questo tutorial descrive il processo di configurazione attraverso diversi scenari e modelli di accesso ai dati reali.

Configurazione delle tabelle DynamoDB

Per iniziare questo tutorial, devi prima seguire i passaggi seguenti per fornire AWS le risorse.

1. Esegui il provisioning AWS delle risorse utilizzando il seguente AWS CloudFormation modello nella CLI:

```
aws cloudformation create-stack \  
  --stack-name AWSAppSyncTutorialForAmazonDynamoDB \  
  --template-url https://s3.us-west-2.amazonaws.com/awsappsync/resources/  
dynamodb/AmazonDynamoDBCFTemplate.yaml \  
  --capabilities CAPABILITY_NAMED_IAM
```

In alternativa, puoi avviare il seguente CloudFormation stack nella regione US-West 2 (Oregon) del tuo account. AWS



Ciò crea quanto segue:

- Una tabella DynamoDB AppSyncTutorial-Post chiamata che conterrà i dati. Post
 - Un ruolo IAM e una policy gestita da IAM associata AWS AppSync per consentire l'interazione con la Post tabella.
2. Per ulteriori informazioni sullo stack e sulle risorse create, esegui il comando dell'interfaccia a riga di comando seguente:

```
aws cloudformation describe-stacks --stack-name AWSAppSyncTutorialForAmazonDynamoDB
```

3. Per eliminare le risorse in un secondo momento, puoi eseguire quanto segue:

```
aws cloudformation delete-stack --stack-name AWSAppSyncTutorialForAmazonDynamoDB
```

Creazione dell'API GraphQL

Per creare l'API GraphQL in: AWS AppSync

1. Accedi a Console di gestione AWS e apri la [AppSync console](#).
 - Nella APIs dashboard, scegli Crea API.
2. Nella finestra Personalizza la tua API o importa da Amazon DynamoDB, scegli Crea da zero.
 - Scegli Start a destra della stessa finestra.
3. Nel campo Nome API, imposta il nome dell'API suAWSAppSyncTutorial.
4. Scegli Create (Crea).

La AWS AppSync console crea una nuova API GraphQL utilizzando la modalità di autenticazione della chiave API. Puoi usare la console per configurare ulteriormente l'API GraphQL ed eseguire query sull'API per le parti restanti di questo tutorial.

Definizione di una post API di base

Ora che hai creato un'API AWS AppSync GraphQL, puoi configurare uno schema di base che consente la creazione, il recupero e l'eliminazione di base dei dati post.

1. [Accedi a Console di gestione AWS e apri la console. AppSync](#)

- Nella APIs dashboard, scegli l'API che hai appena creato.

2. Nella barra laterale, scegli Schema.

- Nel riquadro Schema, sostituisci il contenuto con il seguente codice:

```
schema {
  query: Query
  mutation: Mutation
}

type Query {
  getPost(id: ID!): Post
}

type Mutation {
  addPost(
    id: ID!
    author: String!
    title: String!
    content: String!
    url: String!
  ): Post!
}

type Post {
  id: ID!
  author: String
  title: String
  content: String
  url: String
  ups: Int!
  downs: Int!
  version: Int!
}
```

3. Scegli Save (Salva).

Questo schema definisce un tipo `Post` e le operazioni per aggiungere e ottenere oggetti `Post`.

Configurazione dell'origine dati per le tabelle DynamoDB

Successivamente, collega le query e le mutazioni definite nello schema alla tabella `DynamoDBAppSyncTutorial-Post`.

Innanzitutto, è AWS AppSync necessario conoscere le proprie tabelle. A questo scopo, devi configurare un'origine dati in AWS AppSync:

1. Accedi a Console di gestione AWS e apri la [AppSync console](#).
 - a. Nella APIs dashboard, scegli la tua API GraphQL.
 - b. Nella barra laterale, scegli Data Sources.
2. Seleziona Create data source (Crea origine dati).
 - a. Per il nome dell'origine dati, inserisci `PostDynamoDBTable`
 - b. Per il tipo di origine dati, scegli la tabella Amazon DynamoDB.
 - c. Per Regione, scegli US-WEST-2.
 - d. Per Nome tabella, scegli la tabella `AppSyncTutorial-Post` DynamoDB.
 - e. Crea un nuovo ruolo IAM (consigliato) o scegli un ruolo esistente con `lambda:invokeFunction` autorizzazione IAM. I ruoli esistenti richiedono una policy di fiducia, come spiegato nella sezione [Allegare una fonte di dati](#).

Di seguito è riportato un esempio di policy IAM che dispone delle autorizzazioni necessarie per eseguire operazioni sulla risorsa:

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [ "lambda:invokeFunction" ],
      "Resource": [
        "arn:aws:lambda:us-east-1:111122223333:function:myFunction",
```

```
"arn:aws:lambda:us-east-1:111122223333:function:myFunction:*"
    ]
  }
}
}
```

3. Scegli Create (Crea).

Configurazione del resolver AddPost (DynamoDB) PutItem

Dopo aver AWS AppSync acquisito conoscenza della tabella DynamoDB, è possibile collegarla a singole query e mutazioni definendo i Resolver. Il primo resolver creato è il addPost resolver, che consente di creare un post nella tabella DynamoDB. AppSyncTutorial-Post

Un resolver include i componenti seguenti:

- Posizione nello schema GraphQL per collegare il resolver. In questo caso, stai configurando un resolver nel campo addPost nel tipo Mutation. Questo resolver verrà richiamato quando il chiamante chiama mutation { addPost(...){...} }.
- Origine dati da usare per il resolver. In questo caso, userai l'origine dati PostDynamoDBTable che hai definito in precedenza, per poter aggiungere voci nella tabella DynamoDB AppSyncTutorial-Post.
- Modello di mappatura della richiesta. Lo scopo del modello di mappatura delle richieste è prendere la richiesta in arrivo dal chiamante e tradurla in istruzioni da eseguire con AWS AppSync DynamoDB.
- Modello di mappatura della risposta. Lo scopo del modello di mappatura della risposta è ritradurre la risposta proveniente da DynamoDB nel risultato previsto da GraphQL. Questo è utile se la forma dei dati in DynamoDB è diversa rispetto al tipo Post in GraphQL, ma in questo caso hanno la stessa forma e di conseguenza puoi semplicemente passare i dati.

Per configurare il resolver:

1. [Accedi a e apri la console. Console di gestione AWS AppSync](#)
 - a. Nella APIs dashboard, scegli la tua API GraphQL.
 - b. Nella barra laterale, scegli Data Sources.
2. Seleziona Create data source (Crea origine dati).

- a. Per il nome dell'origine dati, inserisci. `PostDynamoDBTable`
- b. Per il tipo di origine dati, scegli la tabella Amazon DynamoDB.
- c. Per Regione, scegli `US-WEST-2`.
- d. Per Nome tabella, scegli la tabella `AppSyncTutorial-Post DynamoDB`.
- e. Crea un nuovo ruolo IAM (consigliato) o scegli un ruolo esistente con `lamba:invokeFunction` autorizzazione IAM. I ruoli esistenti richiedono una policy di fiducia, come spiegato nella sezione [Allegare una fonte di dati](#).

Di seguito è riportato un esempio di policy IAM che dispone delle autorizzazioni necessarie per eseguire operazioni sulla risorsa:

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [ "lambda:invokeFunction" ],
      "Resource": [
        "arn:aws:lambda:us-west-2:123456789012:function:myFunction",
        "arn:aws:lambda:us-west-2:123456789012:function:myFunction:*"
      ]
    }
  ]
}
```

3. Scegli `Create` (Crea).
4. Scegli la scheda `Schema`.
5. Nel riquadro `Data types` (Tipi di dati a destra, individua il campo `addPost` sul tipo `Mutation` (Mutazione), quindi scegli `Attach` (Collega).
6. Nel menu `Azione`, scegli `Update runtime`, quindi scegli `Unit Resolver` (solo VTL).
7. Nel nome origine dati, scegliere `PostDynamoDBTable`.
8. Incollare il codice seguente nella sezione `Configure the request mapping template` (Configura il modello di mappatura della richiesta):


```
{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key" : {
    "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
  },
  "attributeValues" : {
    "author" : $util.dynamodb.toDynamoDBJson($context.arguments.author),
    "title" : $util.dynamodb.toDynamoDBJson($context.arguments.title),
    "content" : $util.dynamodb.toDynamoDBJson($context.arguments.content),
    "url" : $util.dynamodb.toDynamoDBJson($context.arguments.url),
    "ups" : { "N" : 1 },
    "downs" : { "N" : 0 },
    "version" : { "N" : 1 }
  }
}
```

Nota: un tipo è specificato in tutte le chiavi e in tutti i valori di attributo. Ad esempio, puoi impostare il campo `author` su `{ "S" : "${context.arguments.author}" }`. La `S` parte indica a AWS AppSync e DynamoDB che il valore sarà un valore stringa. Il valore effettivo viene popolato dall'argomento `author`. Analogamente, il campo `version` è un campo numerico, perché usa `N` per il tipo. Infine, stai anche inizializzando i campi `ups`, `downs` e `version`.

Per questo tutorial avete specificato che il `ID!` tipo GraphQL, che indicizza il nuovo elemento inserito in DynamoDB, fa parte degli argomenti del client. AWS AppSync viene fornito con un'utilità per la generazione automatica di ID chiamata `$utils.autoId()` che avresti potuto utilizzare anche sotto forma di `"id" : { "S" : "${utils.autoId()}" }`. Puoi quindi lasciare semplicemente `id: ID!` fuori dalla definizione dello schema `addPost()` e questo verrà inserito automaticamente. Non utilizzerai questa tecnica per questo tutorial, ma dovresti considerarla una buona pratica quando scrivi su tabelle DynamoDB.

Per ulteriori informazioni sui modelli di mappatura, consulta la documentazione di riferimento [Panoramica sui modelli di mappatura dei resolver](#). Per ulteriori informazioni sulla mappatura delle `GetItem` richieste, consulta la [GetItem](#) documentazione di riferimento. Per ulteriori informazioni sui tipi, consulta la documentazione di riferimento [Sistema di tipi \(mappatura della richiesta\)](#).

9. Incollare il codice seguente nella sezione `Configure the request mapping template` (Configura il modello di mappatura della richiesta):

```
$utils.toJson($context.result)
```

Nota: poiché la forma dei dati nella tabella `AppSyncTutorial-Post` corrisponde esattamente alla forma del tipo `Post` in GraphQL, il modello di mappatura della risposta si limita a passare i risultati direttamente. Inoltre, tutti gli esempi mostrati in questo tutorial usano lo stesso modello di mappatura della risposta e di conseguenza devi creare solo un file.

10. Scegli **Save** (Salva).

Chiamata dell'API per aggiungere un post

Ora che il resolver è configurato, AWS AppSync puoi tradurre una `addPost` mutazione in entrata in un'operazione DynamoDB. `PutItem` Puoi ora eseguire una mutazione per inserire contenuto nella tabella.

- Seleziona la scheda **Queries (Query)**.
- Incollare la mutazione seguente nel riquadro **Queries (Query)**.

```
mutation addPost {  
  addPost(  
    id: 123  
    author: "AUTHORNAME"  
    title: "Our first post!"  
    content: "This is our first post."  
    url: "https://aws.amazon.com/appsync/"  
  ) {  
    id  
    author  
    title  
    content  
    url  
    ups  
    downs  
    version  
  }  
}
```

- Scegliere **Execute query** (Esegui query) (il pulsante di riproduzione arancione).

- I risultati del nuovo post creato verranno visualizzati nel riquadro dei risultati a destra del riquadro della query. La schermata visualizzata dovrebbe risultare simile a quella nell'immagine seguente:

```
{
  "data": {
    "addPost": {
      "id": "123",
      "author": "AUTHORNAME",
      "title": "Our first post!",
      "content": "This is our first post.",
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 1
    }
  }
}
```

Ecco che cosa è successo:

- AWS AppSync addPost ha ricevuto una richiesta di mutazione.
- AWS AppSync ha preso la richiesta e il modello di mappatura della richiesta e ha generato un documento di mappatura della richiesta. L'aspetto sarà simile al seguente:

```
{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key" : {
    "id" : { "S" : "123" }
  },
  "attributeValues" : {
    "author": { "S" : "AUTHORNAME" },
    "title": { "S" : "Our first post!" },
    "content": { "S" : "This is our first post." },
    "url": { "S" : "https://aws.amazon.com/appsync/" },
    "ups" : { "N" : 1 },
    "downs" : { "N" : 0 },
    "version" : { "N" : 1 }
  }
}
```

- AWS AppSync ha utilizzato il documento di mappatura della richiesta per generare ed eseguire una richiesta `DynamoDBPutItem`.
- AWS AppSync ha preso i risultati della `PutItem` richiesta e li ha riconvertiti in tipi GraphQL.

```
{
  "id" : "123",
  "author": "AUTHORNAME",
  "title": "Our first post!",
  "content": "This is our first post.",
  "url": "https://aws.amazon.com/appsync/",
  "ups" : 1,
  "downs" : 0,
  "version" : 1
}
```

- Ha passato il documento di mappatura della risposta, che è stato passato invariato.
- Ha restituito il nuovo oggetto creato nella risposta GraphQL.

Configurazione del GetPost Resolver (DynamoDB) GetItem

Ora che sei in grado di aggiungere dati alla tabella `AppSyncTutorial-Post` DynamoDB, devi configurare `getPost` la query in modo che possa recuperare i dati dalla tabella. `AppSyncTutorial-Post` A questo scopo, devi configurare un altro resolver.

- Scegli la scheda Schema.
- Nel riquadro Data types (Tipi di dati) a destra, individuare il campo `getPost` sul tipo Query, quindi scegliere Attach (Collega).
- Nel menu Azione, scegli Update runtime, quindi scegli Unit Resolver (solo VTL).
- Nel nome origine dati, scegliere `PostDynamoDBTable`.
- Incollare il codice seguente nella sezione Configure the request mapping template (Configura il modello di mappatura della richiesta):

```
{
  "version" : "2017-02-28",
  "operation" : "GetItem",
  "key" : {
    "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
  }
}
```

```
}
```

- Incollare il codice seguente nella sezione Configure the request mapping template (Configura il modello di mappatura della richiesta):

```
$utils.toJson($context.result)
```

- Scegli Save (Salva).

Chiamata dell'API per ottenere un post

Ora il resolver è stato configurato, AWS AppSync sa come tradurre una `getPost` query in entrata in un'operazione DynamoDB. `GetItem` Puoi ora eseguire una query per recuperare il post creato prima.

- Seleziona la scheda Queries (Query).
- Nel riquadro Queries (Query) incollare quanto segue:

```
query getPost {  
  getPost(id:123) {  
    id  
    author  
    title  
    content  
    url  
    ups  
    downs  
    version  
  }  
}
```

- Scegliere Execute query (Esegui query) (il pulsante di riproduzione arancione).
- Il post recuperato da DynamoDB dovrebbe apparire nel riquadro dei risultati a destra del riquadro delle query. La schermata visualizzata dovrebbe risultare simile a quella nell'immagine seguente:

```
{  
  "data": {  
    "getPost": {  
      "id": "123",  
      "author": "AUTHORNAME",  
      "title": "Our first post!",
```

```

    "content": "This is our first post.",
    "url": "https://aws.amazon.com/appsync/",
    "ups": 1,
    "downs": 0,
    "version": 1
  }
}
}

```

Ecco che cosa è successo:

- AWS AppSync ha ricevuto una `getPost` richiesta di interrogazione.
- AWS AppSync ha preso la richiesta e il modello di mappatura della richiesta e ha generato un documento di mappatura della richiesta. L'aspetto sarà simile al seguente:

```

{
  "version" : "2017-02-28",
  "operation" : "GetItem",
  "key" : {
    "id" : { "S" : "123" }
  }
}

```

- AWS AppSync ha utilizzato il documento di mappatura della richiesta per generare ed eseguire una richiesta `DynamoDB GetItem`.
- AWS AppSync ha preso i risultati della `GetItem` richiesta e li ha riconvertiti in tipi GraphQL.

```

{
  "id" : "123",
  "author": "AUTHORNAME",
  "title": "Our first post!",
  "content": "This is our first post.",
  "url": "https://aws.amazon.com/appsync/",
  "ups" : 1,
  "downs" : 0,
  "version" : 1
}

```

- Ha passato il documento di mappatura della risposta, che è stato passato invariato.
- Ha restituito l'oggetto recuperato nella risposta.

In alternativa, prendiamo il seguente esempio:

```
query getPost {  
  getPost(id:123) {  
    id  
    author  
    title  
  }  
}
```

Se la tua `getPost` query richiede solo `id`, e `author` `title`, puoi modificare il modello di mappatura della richiesta per utilizzare le espressioni di proiezione per specificare solo gli attributi che desideri dalla tabella DynamoDB per evitare trasferimenti di dati non necessari da DynamoDB a AWS AppSync. Ad esempio, il modello di mappatura della richiesta può essere simile allo snippet riportato di seguito:

```
{  
  "version" : "2017-02-28",  
  "operation" : "GetItem",  
  "key" : {  
    "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)  
  },  
  "projection" : {  
    "expression" : "#author, id, title",  
    "expressionNames" : { "#author" : "author"  
  }  
}
```

Creare una mutazione UpdatePost (DynamoDB) UpdateItem

Finora è possibile creare e recuperare Post oggetti in DynamoDB. Ora configurerai una nuova mutazione per poter aggiornare gli oggetti. Questa operazione verrà eseguita utilizzando l'operazione `UpdateItem` DynamoDB.

- Scegli la scheda Schema.
- Nel riquadro Schema, modificare il tipo `Mutation` per aggiungere una nuova mutazione `updatePost` come segue:

```
type Mutation {  
  updatePost(  

```

```

        id: ID!,
        author: String!,
        title: String!,
        content: String!,
        url: String!
    ): Post
    addPost(
        author: String!
        title: String!
        content: String!
        url: String!
    ): Post!
}

```

- Scegli Save (Salva).
- Nel riquadro Data types (Tipi di dati) a destra, individuare il campo appena creato updatePost sul tipo Mutation (Mutazione), quindi scegliere Attach (Collega).
- Nel menu Azione, scegli Update runtime, quindi scegli Unit Resolver (solo VTL).
- Nel nome origine dati, scegliere PostDynamoDBTable.
- Incollare il codice seguente nella sezione Configure the request mapping template (Configura il modello di mappatura della richiesta):

```

{
  "version" : "2017-02-28",
  "operation" : "UpdateItem",
  "key" : {
    "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
  },
  "update" : {
    "expression" : "SET author = :author, title = :title, content = :content,
#url = :url ADD version :one",
    "expressionNames": {
      "#url" : "url"
    },
    "expressionValues": {
      ":author" : $util.dynamodb.toDynamoDBJson($context.arguments.author),
      ":title" : $util.dynamodb.toDynamoDBJson($context.arguments.title),
      ":content" : $util.dynamodb.toDynamoDBJson($context.arguments.content),
      ":url" : $util.dynamodb.toDynamoDBJson($context.arguments.url),
      ":one" : { "N": 1 }
    }
  }
}

```



```
}  
}
```

Nota: questo resolver utilizza `DynamoDB UpdateItem`, che è significativamente diverso dall'operazione `PutItem`. Invece di scrivere l'intero elemento, stai semplicemente chiedendo a `DynamoDB` di aggiornare determinati attributi. Questa operazione viene eseguita utilizzando `DynamoDB Update Expressions`. L'espressione stessa viene specificata nel campo `expression` nella sezione `update`. L'espressione indica di impostare gli attributi `author`, `title`, `content` e `url` e quindi di incrementare il campo `version`. I valori da usare non appaiono nell'espressione stessa. L'espressione include segnaposto che indicano nomi che iniziano con un carattere di due punti e che vengono quindi definiti nel campo `expressionValues`. Infine, `DynamoDB` contiene parole riservate che non possono apparire in `expression`. Ad esempio, poiché `url` è una parola riservata, per aggiornare il campo `url` puoi usare i segnaposto dei nomi e definirli nel campo `expressionNames`.

Per ulteriori informazioni sulla mappatura delle `UpdateItem` richieste, consulta la [UpdateItem](#) documentazione di riferimento. Per ulteriori informazioni su come scrivere espressioni di aggiornamento, consulta la documentazione di [UpdateExpressions DynamoDB](#).

- Incollare il codice seguente nella sezione `Configure the request mapping template` (Configura il modello di mappatura della richiesta):

```
$utils.toJson($context.result)
```

Chiamata dell'API per aggiornare un post

Ora che il resolver è stato configurato, `AWS AppSync` sa come tradurre una `update` mutazione in entrata in un'operazione `DynamoDB`. `Update` Possiamo ora eseguire una mutazione per aggiornare l'item scritto in precedenza.

- Seleziona la scheda `Queries (Query)`.
- Nel riquadro `Queries (Query)` incollare la seguente mutazione. Dovremo anche aggiornare l'argomento `id` sul valore che abbiamo annotato in precedenza.

```
mutation updatePost {  
  updatePost(  
    id: "123"  
    author: "A new author"  
  )  
}
```

```
    title: "An updated author!"
    content: "Now with updated content!"
    url: "https://aws.amazon.com/appsync/"
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- Scegliere Execute query (Esegui query) (il pulsante di riproduzione arancione).
- Il post aggiornato in DynamoDB dovrebbe apparire nel riquadro dei risultati a destra del riquadro delle query. La schermata visualizzata dovrebbe risultare simile a quella nell'immagine seguente:

```
{
  "data": {
    "updatePost": {
      "id": "123",
      "author": "A new author",
      "title": "An updated author!",
      "content": "Now with updated content!",
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 2
    }
  }
}
```

In questo esempio, i campi `ups` and `downs` non sono stati modificati perché il modello di mappatura delle richieste non AWS AppSync chiedeva a DynamoDB di fare nulla con quei campi. Inoltre, il campo `version` è stato incrementato di 1 perché hai chiesto AWS AppSync a DynamoDB di aggiungere 1 al campo `version`.

Modifica del Resolver UpdatePost (DynamoDB) UpdateItem

Questo è un buon punto di partenza per la mutazione updatePost, ma presenta due problemi principali:

- Se vogliamo aggiornare un solo campo, dovremo aggiornare tutti i campi.
- Se due persone stanno modificando l'oggetto, rischiamo di perdere informazioni.

Per risolvere questi problemi, modificherai la mutazione updatePost in modo da modificare solo gli argomenti che sono stati specificati nella richiesta e quindi aggiungerai una condizione all'operazione UpdateItem.

1. Scegli la scheda Schema.
2. Nel riquadro Schema, modifica il updatePost campo nel Mutation tipo per rimuovere i punti esclamativi dagli url argomentiauthor, titlecontent, e assicurandoti di lasciare il campo così com'è. id In questo modo, questo argomento verrà reso facoltativo. Aggiungere inoltre un nuovo argomento expectedVersion obbligatorio.

```
type Mutation {  
  updatePost(  
    id: ID!,  
    author: String,  
    title: String,  
    content: String,  
    url: String,  
    expectedVersion: Int!  
  ): Post  
  addPost(  
    author: String!  
    title: String!  
    content: String!  
    url: String!  
  ): Post!  
}
```

3. Scegli Save (Salva).
4. Nel riquadro Data types (Tipi di dati) alla destra, individuare il campo updatePost sul tipo Mutation (Mutazione).
5. Scegli PostDynamoDBTable di aprire il resolver esistente.

6. Modifica il modello di mappatura della richiesta nella sezione Configure the request mapping template (Configura il modello di mappatura della richiesta):

```
{
  "version" : "2017-02-28",
  "operation" : "UpdateItem",
  "key" : {
    "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
  },

  ## Set up some space to keep track of things you're updating **
  #set( $expNames = {} )
  #set( $expValues = {} )
  #set( $expSet = {} )
  #set( $expAdd = {} )
  #set( $expRemove = [] )

  ## Increment "version" by 1 **
  ${expAdd.put("version", ":one")}
  ${expValues.put(":one", { "N" : 1 })}

  ## Iterate through each argument, skipping "id" and "expectedVersion" **
  #foreach( $entry in $context.arguments.entrySet() )
    #if( $entry.key != "id" && $entry.key != "expectedVersion" )
      #if( (!$entry.value) && ("${entry.value}" == "") )
        ## If the argument is set to "null", then remove that attribute from
        the item in DynamoDB **

        #set( $discard = ${expRemove.add("#${entry.key}")} )
        ${expNames.put("#${entry.key}", "${entry.key}")}
      #else
        ## Otherwise set (or update) the attribute on the item in DynamoDB **

        ${expSet.put("#${entry.key}", ":${entry.key}")}
        ${expNames.put("#${entry.key}", "${entry.key}")}
        ${expValues.put(":${entry.key}", { "S" : "${entry.value}" })}
      #end
    #end
  #end

  ## Start building the update expression, starting with attributes you're going to
  SET **
  #set( $expression = "" )
```

```

    #if( !${expSet.isEmpty()} )
        #set( $expression = "SET" )
        #foreach( $entry in $expSet.entrySet() )
            #set( $expression = "${expression} ${entry.key} = ${entry.value}" )
            #if ( $foreach.hasNext )
                #set( $expression = "${expression}," )
            #end
        #end
    #end

    ## Continue building the update expression, adding attributes you're going to ADD
    **
    #if( !${expAdd.isEmpty()} )
        #set( $expression = "${expression} ADD" )
        #foreach( $entry in $expAdd.entrySet() )
            #set( $expression = "${expression} ${entry.key} ${entry.value}" )
            #if ( $foreach.hasNext )
                #set( $expression = "${expression}," )
            #end
        #end
    #end

    ## Continue building the update expression, adding attributes you're going to
    REMOVE **
    #if( !${expRemove.isEmpty()} )
        #set( $expression = "${expression} REMOVE" )

        #foreach( $entry in $expRemove )
            #set( $expression = "${expression} ${entry}" )
            #if ( $foreach.hasNext )
                #set( $expression = "${expression}," )
            #end
        #end
    #end

    ## Finally, write the update expression into the document, along with any
    expressionNames and expressionValues **
    "update" : {
        "expression" : "${expression}"
        #if( !${expNames.isEmpty()} )
            ,"expressionNames" : $utils.toJson($expNames)
        #end
        #if( !${expValues.isEmpty()} )
            ,"expressionValues" : $utils.toJson($expValues)
        #end
    }

```

```
        #end
    },

    "condition" : {
        "expression" : "version = :expectedVersion",
        "expressionValues" : {
            ":expectedVersion" :
$util.dynamodb.toDynamoDBJson($context.arguments.expectedVersion)
        }
    }
}
```

7. Scegli Save (Salva).

Questo modello è uno degli esempi più complessi. Dimostra la potenza e la flessibilità dei modelli di mappatura. Passa su tutti gli argomenti, ignorando `id` e `expectedVersion`. Se l'argomento è impostato su qualcosa, chiede AWS AppSync a DynamoDB di aggiornare quell'attributo sull'oggetto in DynamoDB. Se l'attributo è impostato su `null`, chiede AWS AppSync a DynamoDB di rimuovere quell'attributo dall'oggetto post. Se non è stato specificato un argomento, lo ignorerà. Il modello incrementa anche il campo `version`.

È presente anche una nuova sezione `condition`. Un'espressione di condizione consente di indicare AWS AppSync a DynamoDB se la richiesta deve avere successo o meno in base allo stato dell'oggetto già in DynamoDB prima dell'esecuzione dell'operazione. In questo caso, si desidera che la `UpdateItem` richiesta abbia esito positivo solo se il `version` campo dell'elemento attualmente in DynamoDB corrisponde esattamente all'argomento. `expectedVersion`

Per ulteriori informazioni sulle espressioni di condizione, consulta la documentazione di riferimento delle [espressioni di condizione](#).

Chiamata dell'API per aggiornare un post

Proviamo ad aggiornare l'oggetto Post con il nuovo resolver:

- Seleziona la scheda Queries (Query).
- Incollare la mutazione seguente nel riquadro Queries (Query). Dovremo anche aggiornare l'argomento `id` sul valore che abbiamo annotato in precedenza.

```
mutation updatePost {
  updatePost(
    id:123
```

```
    title: "An empty story"
    content: null
    expectedVersion: 2
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- Scegliere Execute query (Esegui query) (il pulsante di riproduzione arancione).
- Il post aggiornato in DynamoDB dovrebbe apparire nel riquadro dei risultati a destra del riquadro delle query. La schermata visualizzata dovrebbe risultare simile a quella nell'immagine seguente:

```
{
  "data": {
    "updatePost": {
      "id": "123",
      "author": "A new author",
      "title": "An empty story",
      "content": null,
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 3
    }
  }
}
```

In questa richiesta, hai chiesto a DynamoDB di aggiornare solo `title` il `content` campo AWS AppSync `and`. Tutti gli altri campi sono stati ignorati (con l'eccezione dell'aumento del campo `version`). Hai impostato l'attributo `title` su un nuovo valore e abbiamo rimosso l'attributo `content` dal post. I campi `author`, `url`, `ups` e `downs` sono stati lasciati invariati.

Prova a eseguire di nuovo la richiesta di mutazione, lasciandola esattamente così com'è. Noterai una risposta simile alla seguente:

```
{
  "data": {
    "updatePost": null
  },
  "errors": [
    {
      "path": [
        "updatePost"
      ],
      "data": {
        "id": "123",
        "author": "A new author",
        "title": "An empty story",
        "content": null,
        "url": "https://aws.amazon.com/appsync/",
        "ups": 1,
        "downs": 0,
        "version": 3
      },
      "errorType": "DynamoDB:ConditionalCheckFailedException",
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ],
      "message": "The conditional request failed (Service: AmazonDynamoDBv2;
Status Code: 400; Error Code: ConditionalCheckFailedException; Request ID:
ABCDEFGHIJKLMN0PQRSTUVWXYZABCDEFGHIJKLMN0PQRSTUVWXYZ)"
    }
  ]
}
```

La richiesta non riesce perché l'espressione di condizione restituisce false:

- La prima volta che hai eseguito la richiesta, il valore del `version` campo del post in DynamoDB 2 era corrispondente all'argomento. `expectedVersion` La richiesta è riuscita, il che significa che il `version` campo è stato incrementato in DynamoDB a. 3
- La seconda volta che hai eseguito la richiesta, il valore del `version` campo del post in DynamoDB 3 era, che non corrispondeva all'argomento. `expectedVersion`

Questo modello viene in genere chiamato blocco ottimistico.

Una caratteristica di un resolver AWS AppSync DynamoDB è che restituisce il valore corrente dell'oggetto post in DynamoDB. Puoi trovare questo valore nel campo `data` nella sezione `errors` della risposta GraphQL. L'applicazione può usare queste informazioni per decidere come procedere. In questo caso, puoi vedere che il `version` campo dell'oggetto in DynamoDB è impostato su, quindi puoi semplicemente aggiornare `expectedVersion` l'argomento 3 3 a e la richiesta avrà nuovamente esito positivo.

Per ulteriori informazioni sulla gestione degli errori del controllo delle condizioni, consulta la documentazione di riferimento dei modelli di mappatura delle [espressioni di condizione](#).

Crea mutazioni UpvotePost e DownvotePost (DynamoDB) UpdateItem

Il tipo `Post` include i campi `ups` e `downs` per consentire di registrare voti positivi e voti negativi, ma finora l'API non ci ha permesso di eseguire alcuna operazione con questi campi. Aggiungiamo ora alcune mutazioni per poter assegnare voti positivi e negativi ai post.

- Scegli la scheda Schema.
- Nel riquadro Schema, modifica il tipo per aggiungere nuove e mutazioni come segue `Mutation`:
`upvotePost` `downvotePost`

```
type Mutation {  
  upvotePost(id: ID!): Post  
  downvotePost(id: ID!): Post  
  updatePost(  
    id: ID!,  
    author: String,  
    title: String,  
    content: String,  
    url: String,  
    expectedVersion: Int!  
  ): Post  
  addPost(  
    author: String!,  
    title: String!,  
    content: String!,  
    url: String!  
  ): Post!  
}
```

- Scegli Save (Salva).
- Nel riquadro Data types (Tipi di dati) a destra, individuare il campo appena creato upvotePost sul tipo Mutation (Mutazione), quindi scegliere Attach (Collega).
- Nel menu Azione, scegli Update runtime, quindi scegli Unit Resolver (solo VTL).
- Nel nome origine dati, scegliere PostDynamoDBTable.
- Incollare il codice seguente nella sezione Configure the request mapping template (Configura il modello di mappatura della richiesta):

```
{
  "version" : "2017-02-28",
  "operation" : "UpdateItem",
  "key" : {
    "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
  },
  "update" : {
    "expression" : "ADD ups :plusOne, version :plusOne",
    "expressionValues" : {
      ":plusOne" : { "N" : 1 }
    }
  }
}
```

- Incollare il codice seguente nella sezione Configure the request mapping template (Configura il modello di mappatura della richiesta):

```
$utils.toJson($context.result)
```

- Scegli Save (Salva).
- Nel riquadro Data types (Tipi di dati) a destra, individuare il campo downvotePost appena creato sul tipo Mutation (Mutazione), quindi scegliere Attach (Collega).
- Nel nome origine dati, scegliere PostDynamoDBTable.
- Incollare il codice seguente nella sezione Configure the request mapping template (Configura il modello di mappatura della richiesta):

```
{
  "version" : "2017-02-28",
  "operation" : "UpdateItem",
  "key" : {
    "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
```

```

    },
    "update" : {
      "expression" : "ADD downs :plusOne, version :plusOne",
      "expressionValues" : {
        ":plusOne" : { "N" : 1 }
      }
    }
  }
}

```

- Incollare il codice seguente nella sezione Configure the request mapping template (Configura il modello di mappatura della richiesta):

```
$utils.toJson($context.result)
```

- Scegli Save (Salva).

Chiamata dell'API per assegnare voti positivi e voti negativi a un post

Ora i nuovi resolver sono stati configurati, AWS AppSync sa come tradurre un'operazione in entrata o una upvotePost mutazione downvote in DynamoDB. UpdateItem Ora puoi eseguire mutazioni per assegnare voti positivi e voti negativi al post creato prima.

- Seleziona la scheda Queries (Query).
- Incollare la mutazione seguente nel riquadro Queries (Query). Dovremo anche aggiornare l'argomento id sul valore che abbiamo annotato in precedenza.

```

mutation votePost {
  upvotePost(id:123) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}

```

- Scegliere Execute query (Esegui query) (il pulsante di riproduzione arancione).

- Il post viene aggiornato in DynamoDB e dovrebbe apparire nel riquadro dei risultati a destra del riquadro delle query. La schermata visualizzata dovrebbe risultare simile a quella nell'immagine seguente:

```
{
  "data": {
    "upvotePost": {
      "id": "123",
      "author": "A new author",
      "title": "An empty story",
      "content": null,
      "url": "https://aws.amazon.com/appsync/",
      "ups": 6,
      "downs": 0,
      "version": 4
    }
  }
}
```

- Scegli **Execute query** (Esegui query) più volte. I campi `ups` e `version` vengono incrementati di 1 ogni volta che esegui la query.
- Modifica la query in modo da chiamare la mutazione `downvotePost` come segue:

```
mutation votePost {
  downvotePost(id:123) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- Scegliere **Execute query** (Esegui query) (il pulsante di riproduzione arancione). Questa volta, i campi `downs` e `version` vengono incrementati di 1 ogni volta che esegui la query.

```
{
  "data": {
    "downvotePost": {
```

```
    "id": "123",
    "author": "A new author",
    "title": "An empty story",
    "content": null,
    "url": "https://aws.amazon.com/appsync/",
    "ups": 6,
    "downs": 4,
    "version": 12
  }
}
```

Configurazione del DeletePost Resolver (DynamoDB) DeleteItem

La prossima mutazione che vuoi configurare è quella per l'eliminazione di un post. Questa operazione verrà eseguita utilizzando l'operazione DeleteItem DynamoDB.

- Scegli la scheda Schema.
- Nel riquadro Schema, modificare il tipo Mutation per aggiungere una nuova mutazione deletePost come segue:

```
type Mutation {
  deletePost(id: ID!, expectedVersion: Int): Post
  upvotePost(id: ID!): Post
  downvotePost(id: ID!): Post
  updatePost(
    id: ID!,
    author: String,
    title: String,
    content: String,
    url: String,
    expectedVersion: Int!
  ): Post
  addPost(
    author: String!,
    title: String!,
    content: String!,
    url: String!
  ): Post!
}
```

A questo punto il campo `expectedVersion` viene reso opzionale, operazione spiegata in seguito al momento di aggiungere il modello di mappatura della richiesta.

- Scegli **Save (Salva)**.
- Nel riquadro **Data types (Tipi di dati)** a destra, individuare il campo appena creato **delete** (elimina) sul tipo **Mutation (Mutazione)**, quindi scegliere **Attach (Collega)**.
- Nel menu **Azione**, scegli **Update runtime**, quindi scegli **Unit Resolver (solo VTL)**.
- Nel nome origine dati, scegliere **PostDynamoDBTable**.
- Incollare il codice seguente nella sezione **Configure the request mapping template (Configura il modello di mappatura della richiesta)**:

```
{
  "version" : "2017-02-28",
  "operation" : "DeleteItem",
  "key": {
    "id": $util.dynamodb.toDynamoDBJson($context.arguments.id)
  }
  #if( $context.arguments.containsKey("expectedVersion") )
    , "condition" : {
      "expression" : "attribute_not_exists(id) OR version
= :expectedVersion",
      "expressionValues" : {
        ":expectedVersion" :
$util.dynamodb.toDynamoDBJson($context.arguments.expectedVersion)
      }
    }
  #end
}
```

Nota: l'argomento `expectedVersion` è un argomento facoltativo. Se il chiamante imposta un `expectedVersion` argomento nella richiesta, il modello aggiunge una condizione che consente l'esito positivo della `DeleteItem` richiesta solo se l'elemento è già stato eliminato o se l'`version` attributo del post in DynamoDB corrisponde esattamente a `expectedVersion`. Se non viene inserito, nessuna espressione di condizione viene specificata nella richiesta `DeleteItem`. Ha successo indipendentemente dal valore o dal fatto che l'elemento esista o meno in DynamoDB.

- Incollare il codice seguente nella sezione **Configure the request mapping template (Configura il modello di mappatura della richiesta)**:

```
$utils.toJson($context.result)
```

Nota: anche se stai eliminando un item, puoi restituire l'item che è stato eliminato, se non è già stato eliminato.

- Scegli Save (Salva).

Per ulteriori informazioni sulla mappatura delle DeleteItem richieste, consulta la documentazione di riferimento. [DeleteItem](#)

Chiamata dell'API per eliminare un post

Ora che il resolver è stato configurato, AWS AppSync sa come tradurre una delete mutazione in entrata in un'operazione DynamoDB. DeleteItem Puoi ora eseguire una mutazione per eliminare qualcosa nella tabella.

- Seleziona la scheda Queries (Query).
- Incollare la mutazione seguente nel riquadro Queries (Query). Dovremo anche aggiornare l'argomento id sul valore che abbiamo annotato in precedenza.

```
mutation deletePost {  
  deletePost(id:123) {  
    id  
    author  
    title  
    content  
    url  
    ups  
    downs  
    version  
  }  
}
```

- Scegliere Execute query (Esegui query) (il pulsante di riproduzione arancione).
- Il post viene eliminato da DynamoDB. Nota che AWS AppSync restituisce il valore dell'elemento che è stato eliminato da DynamoDB, che dovrebbe apparire nel riquadro dei risultati a destra del riquadro delle query. La schermata visualizzata dovrebbe risultare simile a quella nell'immagine seguente:

```
{
  "data": {
    "deletePost": {
      "id": "123",
      "author": "A new author",
      "title": "An empty story",
      "content": null,
      "url": "https://aws.amazon.com/appsync/",
      "ups": 6,
      "downs": 4,
      "version": 12
    }
  }
}
```

Il valore viene restituito solo se la chiamata a `deletePost` è quella che lo ha effettivamente eliminato da DynamoDB.

- Scegliere nuovamente Execute query (Esegui query).
- La chiamata riesce ancora, ma non viene restituito alcun valore.

```
{
  "data": {
    "deletePost": null
  }
}
```

Proviamo ora a eliminare un post, questa volta specificando `expectedValue`. In primo luogo, tuttavia, dovrai creare un nuovo post perché hai appena eliminato quello usato fino a questo punto.

- Incollare la mutazione seguente nel riquadro Queries (Query).

```
mutation addPost {
  addPost(
    id:123
    author: "AUTHORNAME"
    title: "Our second post!"
    content: "A new post."
    url: "https://aws.amazon.com/appsync/"
  )
}
```



```

    ) {
      id
      author
      title
      content
      url
      ups
      downs
      version
    }
  }
}

```

- Scegliere Execute query (Esegui query) (il pulsante di riproduzione arancione).
- I risultati del nuovo post creato verranno visualizzati nel riquadro dei risultati a destra del riquadro della query. Annotare il valore di `id` del nuovo oggetto creato, perché sarà necessario tra poco. La schermata visualizzata dovrebbe risultare simile a quella nell'immagine seguente:

```

{
  "data": {
    "addPost": {
      "id": "123",
      "author": "AUTHORNAME",
      "title": "Our second post!",
      "content": "A new post.",
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 1
    }
  }
}

```

Proviamo ora a eliminare il post, ma inserendo un valore errato per `expectedVersion`:

- Incollare la mutazione seguente nel riquadro Queries (Query). Dovremo anche aggiornare l'argomento `id` sul valore che abbiamo annotato in precedenza.

```

mutation deletePost {
  deletePost(
    id:123
    expectedVersion: 9999
  )
}

```

```

    ) {
      id
      author
      title
      content
      url
      ups
      downs
      version
    }
  }
}

```

- Scegliere **Execute query** (Esegui query) (il pulsante di riproduzione arancione).

```

{
  "data": {
    "deletePost": null
  },
  "errors": [
    {
      "path": [
        "deletePost"
      ],
      "data": {
        "id": "123",
        "author": "AUTHORNAME",
        "title": "Our second post!",
        "content": "A new post.",
        "url": "https://aws.amazon.com/appsync/",
        "ups": 1,
        "downs": 0,
        "version": 1
      },
      "errorType": "DynamoDB:ConditionalCheckFailedException",
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ],
      "message": "The conditional request failed (Service: AmazonDynamoDBv2; Status Code: 400; Error Code: ConditionalCheckFailedException; Request ID: ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ)"
    }
  ]
}

```

```
]
}
```

La richiesta non è riuscita perché l'espressione della condizione risulta falsa: il valore `version` del post in DynamoDB non corrisponde a quello specificato `expectedValue` negli argomenti. Il valore corrente dell'oggetto viene restituito nel campo `data` nella sezione `errors` della risposta GraphQL.

- Riprova la richiesta, correggendo il valore di `expectedVersion`:

```
mutation deletePost {
  deletePost(
    id:123
    expectedVersion: 1
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- Scegliere **Execute query** (Esegui query) (il pulsante di riproduzione arancione).
- Questa volta la richiesta ha esito positivo e viene restituito il valore che è stato eliminato da DynamoDB:

```
{
  "data": {
    "deletePost": {
      "id": "123",
      "author": "AUTHORNAME",
      "title": "Our second post!",
      "content": "A new post.",
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 1
    }
  }
}
```

```
}  
}
```

- Scegliere nuovamente Execute query (Esegui query).
- La chiamata riesce ancora, ma questa volta non viene restituito alcun valore perché il post è già stato eliminato in DynamoDB.

```
{  
  "data": {  
    "deletePost": null  
  }  
}
```

Configurazione del resolver Up the allPost (scansione di DynamoDB)

Fino a questo punto l'API è utile solo se conosci l'id di ogni post che vuoi guardare. Aggiungiamo ora un nuovo resolver che restituisce tutti i post nella tabella.

- Scegli la scheda Schema.
- Nel riquadro Schema, modificare il tipo Query per aggiungere una nuova query allPost come segue:

```
type Query {  
  allPost(count: Int, nextToken: String): PaginatedPosts!  
  getPost(id: ID): Post  
}
```

- Aggiungi un nuovo tipo PaginationPosts:

```
type PaginatedPosts {  
  posts: [Post!]!  
  nextToken: String  
}
```

- Scegli Save (Salva).
- Nel riquadro Data types (Tipi di dati) a destra, individuare il campo appena creato allPost sul tipo Query, quindi scegliere Attach (Collega).
- Nel menu Azione, scegli Update runtime, quindi scegli Unit Resolver (solo VTL).

- Nel nome origine dati, scegliere PostDynamoDBTable.
- Incollare il codice seguente nella sezione Configure the request mapping template (Configura il modello di mappatura della richiesta):

```
{
  "version" : "2017-02-28",
  "operation" : "Scan"
  #if( ${context.arguments.count} )
    , "limit": $util.toJson($context.arguments.count)
  #end
  #if( ${context.arguments.nextToken} )
    , "nextToken": $util.toJson($context.arguments.nextToken)
  #end
}
```

Questo resolver include due argomenti facoltativi: `count`, che specifica il numero massimo di item da restituire in una singola chiamata, e `nextToken`, che può essere usato per recuperare il set successivo di risultati (vedrai da dove proviene il valore per `nextToken` più tardi).

- Incollare il codice seguente nella sezione Configure the request mapping template (Configura il modello di mappatura della richiesta):

```
{
  "posts": $utils.toJson($context.result.items)
  #if( ${context.result.nextToken} )
    , "nextToken": $util.toJson($context.result.nextToken)
  #end
}
```

Nota: questo modello di mappatura della risposta è diverso da tutti gli altri descritti finora. Il risultato della query `allPost` è un oggetto `PaginatedPosts`, che contiene un elenco dei post e un token di paginazione. La forma di questo oggetto è diversa da quella restituita dal Resolver AWS AppSync DynamoDB: l'elenco dei post viene richiamato `items` nei risultati del Resolver AWS AppSync DynamoDB, ma viene richiamato `posts` `PaginatedPosts`.

- Scegli Save (Salva).

Per ulteriori informazioni sulla mappatura della richiesta `Scan`, consulta la documentazione di riferimento di [Scan](#).

Chiamata dell'API per analizzare tutti i post

Ora il resolver è stato configurato, AWS AppSync sa come tradurre una `allPost` query in entrata in un'operazione DynamoDB. Scan Puoi ora analizzare la tabella per recuperare tutti i post.

Prima di provare, devi immettere nella tabella alcuni dati, perché hai eliminato tutti quelli usati finora.

- Seleziona la scheda Queries (Query).
- Incollare la mutazione seguente nel riquadro Queries (Query).

```
mutation addPost {
  post1: addPost(id:1 author: "AUTHORNAME" title: "A series of posts, Volume 1"
  content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post2: addPost(id:2 author: "AUTHORNAME" title: "A series of posts, Volume 2"
  content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post3: addPost(id:3 author: "AUTHORNAME" title: "A series of posts, Volume 3"
  content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post4: addPost(id:4 author: "AUTHORNAME" title: "A series of posts, Volume 4"
  content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post5: addPost(id:5 author: "AUTHORNAME" title: "A series of posts, Volume 5"
  content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post6: addPost(id:6 author: "AUTHORNAME" title: "A series of posts, Volume 6"
  content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post7: addPost(id:7 author: "AUTHORNAME" title: "A series of posts, Volume 7"
  content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post8: addPost(id:8 author: "AUTHORNAME" title: "A series of posts, Volume 8"
  content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post9: addPost(id:9 author: "AUTHORNAME" title: "A series of posts, Volume 9"
  content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
}
```

- Scegliere Execute query (Esegui query) (il pulsante di riproduzione arancione).

Analizziamo ora la tabella, restituendo cinque risultati per volta.

- Incollare la query seguente nel riquadro Queries (Query):

```
query allPost {
  allPost(count: 5) {
    posts {
      id
      title
    }
  }
}
```

```

    }
    nextToken
  }
}

```

- Scegliere Execute query (Esegui query) (il pulsante di riproduzione arancione).
- I primi cinque post appariranno nel riquadro dei risultati a destra del riquadro della query. La schermata visualizzata dovrebbe risultare simile a quella nell'immagine seguente:

```

{
  "data": {
    "allPost": {
      "posts": [
        {
          "id": "5",
          "title": "A series of posts, Volume 5"
        },
        {
          "id": "1",
          "title": "A series of posts, Volume 1"
        },
        {
          "id": "6",
          "title": "A series of posts, Volume 6"
        },
        {
          "id": "9",
          "title": "A series of posts, Volume 9"
        },
        {
          "id": "7",
          "title": "A series of posts, Volume 7"
        }
      ],
      "nextToken":
        "eyJ2ZXJzaW9uIjoxLCJ0b2t1biI6IkFRSUNBSGo4eHR0RG0xWXhUa1F0cEhXMEp1R3B0M1B3eTh0SmRvcG9ad2RHYjI
    }
  }
}

```

Hai ottenuto cinque risultati e una `nextToken` che è possibile utilizzare per ottenere la prossima serie di risultati.

- Aggiornare la query `allPost` in modo da includere `nextToken` dal set precedente di risultati:

```
query allPost {
  allPost(
    count: 5
    nextToken:
"eyJ2ZXJzaW9uIjoxLCJ0b2t1biI6IkFRSUNBSGo4eHR0RG0xWXhUa1F0cEhXMEp1R3B0M1B3eTh0SmRvcG9ad2RHYjI
  ) {
    posts {
      id
      author
    }
    nextToken
  }
}
```

- Scegliere **Execute query** (Esegui query) (il pulsante di riproduzione arancione).
- I quattro post rimanenti appariranno nel riquadro dei risultati a destra del riquadro della query. Questo set di risultati non contiene `nextToken`, perché hai analizzato tutti i nove post, senza che ne sia rimasto alcuno. La schermata visualizzata dovrebbe risultare simile a quella nell'immagine seguente:

```
{
  "data": {
    "allPost": {
      "posts": [
        {
          "id": "2",
          "title": "A series of posts, Volume 2"
        },
        {
          "id": "3",
          "title": "A series of posts, Volume 3"
        },
        {
          "id": "4",
          "title": "A series of posts, Volume 4"
        },
        {
```



```

        "id": "8",
        "title": "A series of posts, Volume 8"
      }
    ],
    "nextToken": null
  }
}

```

Configurazione dell' allPostsByAuthor Resolver (DynamoDB Query)

Oltre a scansionare DynamoDB per tutti i post, puoi anche interrogare DynamoDB per recuperare i post creati da un autore specifico. La tabella DynamoDB che hai creato in precedenza contiene già `GlobalSecondaryIndex` una `author-index` chiamata che puoi utilizzare con un'operazione DynamoDB per recuperare tutti i Query post creati da un autore specifico.

- Scegli la scheda Schema.
- Nel riquadro Schema, modificare il tipo Query per aggiungere una nuova query `allPostsByAuthor` come segue:

```

type Query {
  allPostsByAuthor(author: String!, count: Int, nextToken: String): PaginatedPosts!
  allPost(count: Int, nextToken: String): PaginatedPosts!
  getPost(id: ID): Post
}

```

Nota: usa lo stesso tipo `PaginatedPosts` utilizzato con la query `allPost`.

- Scegli Save (Salva).
- Nel riquadro Tipi di dati a destra, individua il campo `allPostsByAutore` appena creato nel campo Tipo di query, quindi scegli Allega.
- Nel menu Azione, scegli Update runtime, quindi scegli Unit Resolver (solo VTL).
- Nel nome origine dati, scegliere `PostDynamoDBTable`.
- Incollare il codice seguente nella sezione Configure the request mapping template (Configura il modello di mappatura della richiesta):

```

{
  "version" : "2017-02-28",

```

```

    "operation" : "Query",
    "index" : "author-index",
    "query" : {
      "expression": "author = :author",
      "expressionValues" : {
        ":author" : $util.dynamodb.toDynamoDBJson($context.arguments.author)
      }
    }
    #if( ${context.arguments.count} )
      , "limit": $util.toJson($context.arguments.count)
    #end
    #if( ${context.arguments.nextToken} )
      , "nextToken": "${context.arguments.nextToken}"
    #end
  }
}

```

Come per il resolver `allPost`, questo resolver include due argomenti facoltativi: `count`, che specifica il numero massimo di item da restituire in una singola chiamata, e `nextToken`, che può essere usato per recuperare il set successivo di risultati (il valore per `nextToken` può essere ottenuto da una chiamata precedente).

- Incollare il codice seguente nella sezione `Configure the request mapping template` (Configura il modello di mappatura della richiesta):

```

{
  "posts": $utils.toJson($context.result.items)
  #if( ${context.result.nextToken} )
    , "nextToken": $util.toJson($context.result.nextToken)
  #end
}

```

Nota: questo è lo stesso modello di mappatura della risposta usato nel resolver `allPost`.

- Scegli `Save` (Salva).

Per ulteriori informazioni sulla mappatura della richiesta `Query`, consulta la documentazione di riferimento di [Query](#).

Chiamata dell'API per eseguire una query per recuperare tutti i post di un autore

Ora che il resolver è stato configurato, AWS AppSync sa come tradurre una `allPostsByAuthor` mutazione in entrata in un'operazione DynamoDB rispetto all'indice. Query `author-index` Puoi ora eseguire una query sulla tabella per recuperare tutti i post di un autore specifico.

Prima di continuare, tuttavia, devi immettere altri post nella tabella, perché i post presenti fino a questo punto hanno lo stesso autore.

- Seleziona la scheda Queries (Query).
- Incollare la mutazione seguente nel riquadro Queries (Query).

```
mutation addPost {  
  post1: addPost(id:10 author: "Nadia" title: "The cutest dog in the world" content:  
    "So cute. So very, very cute." url: "https://aws.amazon.com/appsync/" ) { author,  
    title }  
  post2: addPost(id:11 author: "Nadia" title: "Did you know...?" content: "AppSync  
    works offline?" url: "https://aws.amazon.com/appsync/" ) { author, title }  
  post3: addPost(id:12 author: "Steve" title: "I like GraphQL" content: "It's great"  
    url: "https://aws.amazon.com/appsync/" ) { author, title }  
}
```

- Scegliere Execute query (Esegui query) (il pulsante di riproduzione arancione).

Esegui ora la query sulla tabella, per restituire tutti i post il cui autore è Nadia.

- Incollare la query seguente nel riquadro Queries (Query):

```
query allPostsByAuthor {  
  allPostsByAuthor(author: "Nadia") {  
    posts {  
      id  
      title  
    }  
    nextToken  
  }  
}
```

- Scegliere Execute query (Esegui query) (il pulsante di riproduzione arancione).
- Tutti i post il cui autore è Nadia verranno visualizzati nel riquadro dei risultati a destra del riquadro della query. La schermata visualizzata dovrebbe risultare simile a quella nell'immagine seguente:

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world"
        },
        {
          "id": "11",
          "title": "Did you know...?"
        }
      ],
      "nextToken": null
    }
  }
}
```

La paginazione funziona per Query esattamente come per Scan. Ad esempio, cerchiamo tutti i post di AUTHORNAME, restituendone cinque per volta.

- Incollare la query seguente nel riquadro Queries (Query):

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "AUTHORNAME"
    count: 5
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

- Scegliere Execute query (Esegui query) (il pulsante di riproduzione arancione).
- Tutti i post il cui autore è AUTHORNAME verranno visualizzati nel riquadro dei risultati a destra del riquadro della query. La schermata visualizzata dovrebbe risultare simile a quella nell'immagine seguente:

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "6",
          "title": "A series of posts, Volume 6"
        },
        {
          "id": "4",
          "title": "A series of posts, Volume 4"
        },
        {
          "id": "2",
          "title": "A series of posts, Volume 2"
        },
        {
          "id": "7",
          "title": "A series of posts, Volume 7"
        },
        {
          "id": "1",
          "title": "A series of posts, Volume 1"
        }
      ],
      "nextToken":
        "eyJ2ZXJzaW9uIjoyLCJ0b2t1biI6IkFRSUNBSGo4eHR0RG0xWXhUa1F0cEhXMEp1R3B0M1B3eTh0SmRvcG9ad2RHYjI"
    }
  }
}
```

- Aggiornare l'argomento `nextToken` con il valore restituito dalla query precedente, come segue:

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "AUTHORNAME"
    count: 5
    nextToken:
      "eyJ2ZXJzaW9uIjoyLCJ0b2t1biI6IkFRSUNBSGo4eHR0RG0xWXhUa1F0cEhXMEp1R3B0M1B3eTh0SmRvcG9ad2RHYjI"
  ) {
    posts {
      id
    }
  }
}
```

```
    title
  }
  nextToken
}
```

- Scegliere Execute query (Esegui query) (il pulsante di riproduzione arancione).
- Tutti i post rimanenti il cui autore è AUTHORNAME verranno visualizzati nel riquadro dei risultati a destra del riquadro della query. La schermata visualizzata dovrebbe risultare simile a quella nell'immagine seguente:

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "8",
          "title": "A series of posts, Volume 8"
        },
        {
          "id": "5",
          "title": "A series of posts, Volume 5"
        },
        {
          "id": "3",
          "title": "A series of posts, Volume 3"
        },
        {
          "id": "9",
          "title": "A series of posts, Volume 9"
        }
      ],
      "nextToken": null
    }
  }
}
```

Uso di set

Fino a questo punto il Post tipo era un oggetto piatto. key/value. È inoltre possibile modellare oggetti complessi con il resolver AWS AppSyncDynamo DB, come set, elenchi e mappe.

Aggiorniamo ora il tipo `Post` perché includa `tag`. Un post può avere 0 o più tag, che vengono memorizzati in DynamoDB come set di stringhe. Dobbiamo anche configurare alcune mutazioni per aggiungere e rimuovere tag e una query per analizzare i post con un tag specifico.

- Scegli la scheda Schema.
- Nel riquadro Schema, modificare il tipo `Post` per aggiungere un nuovo campo `tags` come segue:

```
type Post {  
  id: ID!  
  author: String  
  title: String  
  content: String  
  url: String  
  ups: Int!  
  downs: Int!  
  version: Int!  
  tags: [String!]  
}
```

- Nel riquadro Schema, modificare il tipo `Query` per aggiungere una nuova query `allPostsByTag` come segue:

```
type Query {  
  allPostsByTag(tag: String!, count: Int, nextToken: String): PaginatedPosts!  
  allPostsByAuthor(author: String!, count: Int, nextToken: String): PaginatedPosts!  
  allPost(count: Int, nextToken: String): PaginatedPosts!  
  getPost(id: ID): Post  
}
```

- Nel riquadro Schema, modifica il `Mutation` tipo per aggiungere nuove `addTag` e `removeTag` mutazioni come segue:

```
type Mutation {  
  addTag(id: ID!, tag: String!): Post  
  removeTag(id: ID!, tag: String!): Post  
  deletePost(id: ID!, expectedVersion: Int): Post  
  upvotePost(id: ID!): Post  
  downvotePost(id: ID!): Post  
  updatePost(  
    id: ID!,  
    author: String,  
    title: String,  
  )
```

```

    content: String,
    url: String,
    expectedVersion: Int!
  ): Post
  addPost(
    author: String!,
    title: String!,
    content: String!,
    url: String!
  ): Post!
}
```

- Scegli Save (Salva).
- Nel riquadro Tipi di dati a destra, trova il campo allPostsByTag appena creato nel tipo di query, quindi scegli Allega.
- Nel nome origine dati, scegliere PostDynamoDBTable.
- Incollare il codice seguente nella sezione Configure the request mapping template (Configura il modello di mappatura della richiesta):

```

{
  "version" : "2017-02-28",
  "operation" : "Scan",
  "filter": {
    "expression": "contains (tags, :tag)",
    "expressionValues": {
      ":tag": $util.dynamodb.toDynamoDBJson($context.arguments.tag)
    }
  }
}
#if( ${context.arguments.count} )
  , "limit": $util.toJson($context.arguments.count)
#end
#if( ${context.arguments.nextToken} )
  , "nextToken": $util.toJson($context.arguments.nextToken)
#end
}
```

- Incollare il codice seguente nella sezione Configure the request mapping template (Configura il modello di mappatura della richiesta):

```

{
  "posts": $utils.toJson($context.result.items)
  #if( ${context.result.nextToken} )
```



```

    , "nextToken": $util.toJson($context.result.nextToken)
  #end
}

```

- Scegli Save (Salva).
- Nel riquadro Data types (Tipi di dati) a destra, individuare il campo appena creato addTag sul tipo Mutation (Mutazione), quindi scegliere Attach (Collega).
- Nel nome origine dati, scegliere PostDynamoDBTable.
- Incollare il codice seguente nella sezione Configure the request mapping template (Configura il modello di mappatura della richiesta):

```

{
  "version" : "2017-02-28",
  "operation" : "UpdateItem",
  "key" : {
    "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
  },
  "update" : {
    "expression" : "ADD tags :tags, version :plusOne",
    "expressionValues" : {
      ":tags" : { "SS": [ $util.toJson($context.arguments.tag) ] },
      ":plusOne" : { "N" : 1 }
    }
  }
}

```

- Incollare il codice seguente nella sezione Configure the request mapping template (Configura il modello di mappatura della richiesta):

```

$utils.toJson($context.result)

```

- Scegli Save (Salva).
- Nel riquadro Data types (Tipi di dati) a destra, individuare il campo appena creato removeTag sul tipo Mutation (Mutazione), quindi scegliere Attach (Collega).
- Nel nome origine dati, scegliere PostDynamoDBTable.
- Incollare il codice seguente nella sezione Configure the request mapping template (Configura il modello di mappatura della richiesta):

```

{
  "version" : "2017-02-28",

```

```

    "operation" : "UpdateItem",
    "key" : {
      "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
    },
    "update" : {
      "expression" : "DELETE tags :tags ADD version :plusOne",
      "expressionValues" : {
        ":tags" : { "SS": [ $util.toJson($context.arguments.tag) ] },
        ":plusOne" : { "N" : 1 }
      }
    }
  }
}

```

- Incollare il codice seguente nella sezione Configure the request mapping template (Configura il modello di mappatura della richiesta):

```
$utils.toJson($context.result)
```

- Scegli Save (Salva).

Chiamata dell'API per usare tag

Ora che hai configurato i resolver, AWS AppSync sa come tradurre le richieste in entrata addTag e le richieste allPostsByTag in DynamoDB e nelle operazioni. removeTag UpdateItem Scan

Per provare, selezioniamo uno dei post creati prima. Ad esempio, è possibile utilizzare un post redatto dall'utente Nadia.

- Seleziona la scheda Queries (Query).
- Incollare la query seguente nel riquadro Queries (Query):

```

query allPostsByAuthor {
  allPostsByAuthor(
    author: "Nadia"
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}

```

```
}
```

- Scegliere Execute query (Esegui query) (il pulsante di riproduzione arancione).
- Tutti i post di Nadia verranno visualizzati nel riquadro dei risultati a destra del riquadro della query. La schermata visualizzata dovrebbe risultare simile a quella nell'immagine seguente:

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world"
        },
        {
          "id": "11",
          "title": "Did you known...?"
        }
      ],
      "nextToken": null
    }
  }
}
```

- Utilizza quella con il titolo "The cutest dog in the world". Annotiamo il valore di `id`, perché dovrà essere usato più tardi.

Proviamo ora ad aggiungere un tag dog.

- Incollare la mutazione seguente nel riquadro Queries (Query). Dovremo anche aggiornare l'argomento `id` sul valore che abbiamo annotato in precedenza.

```
mutation addTag {
  addTag(id:10 tag: "dog") {
    id
    title
    tags
  }
}
```

- Scegliere Execute query (Esegui query) (il pulsante di riproduzione arancione).

- Il post viene aggiornato con il nuovo tag.

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog"
      ]
    }
  }
}
```

Possiamo anche aggiungere più tag come segue:

- Aggiornare la mutazione in modo da modificare l'argomento tag in puppy.

```
mutation addTag {
  addTag(id:10 tag: "puppy") {
    id
    title
    tags
  }
}
```

- Scegliere Execute query (Esegui query) (il pulsante di riproduzione arancione).
- Il post viene aggiornato con il nuovo tag.

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog",
        "puppy"
      ]
    }
  }
}
```

Puoi anche eliminare tag:

- Incollare la mutazione seguente nel riquadro Queries (Query). Dovremo anche aggiornare l'argomento `id` sul valore che abbiamo annotato in precedenza.

```
mutation removeTag {  
  removeTag(id:10 tag: "puppy") {  
    id  
    title  
    tags  
  }  
}
```

- Scegliere Execute query (Esegui query) (il pulsante di riproduzione arancione).
- Il post viene aggiornato e il tag puppy viene eliminato.

```
{  
  "data": {  
    "addTag": {  
      "id": "10",  
      "title": "The cutest dog in the world",  
      "tags": [  
        "dog"  
      ]  
    }  
  }  
}
```

Puoi anche cercare tutti i post che includono un tag:

- Incollare la query seguente nel riquadro Queries (Query):

```
query allPostsByTag {  
  allPostsByTag(tag: "dog") {  
    posts {  
      id  
      title  
      tags  
    }  
    nextToken  
  }  
}
```

```
}
```

- Scegliere Execute query (Esegui query) (il pulsante di riproduzione arancione).
- Vengono restituiti tutti i post con il tag dog come segue:

```
{
  "data": {
    "allPostsByTag": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world",
          "tags": [
            "dog",
            "puppy"
          ]
        }
      ],
      "nextToken": null
    }
  }
}
```

Uso di elenchi e mappe

Oltre a utilizzare i set DynamoDB, puoi anche utilizzare elenchi e mappe DynamoDB per modellare dati complessi in un singolo oggetto.

Valutiamo ora la possibilità di aggiungere commenti ai post. Questo verrà modellato come un elenco di oggetti della mappa sull'Post oggetto in DynamoDB.

Nota: in un'applicazione reale, i commenti sarebbero modellati nella propria tabella. Per questo tutorial, aggiungerli nella tabella Post.

- Scegli la scheda Schema.
- Nel riquadro Schema, aggiungere un nuovo tipo Comment come segue:

```
type Comment {
  author: String!
  comment: String!
```

```
}
```

- Nel riquadro Schema, modificare il tipo `Post` per aggiungere un nuovo campo `comments` come segue:

```
type Post {  
  id: ID!  
  author: String  
  title: String  
  content: String  
  url: String  
  ups: Int!  
  downs: Int!  
  version: Int!  
  tags: [String!]  
  comments: [Comment!]  
}
```

- Nel riquadro Schema, modificare il tipo `Mutation` per aggiungere una nuova mutazione `addComment` come segue:

```
type Mutation {  
  addComment(id: ID!, author: String!, comment: String!): Post  
  addTag(id: ID!, tag: String!): Post  
  removeTag(id: ID!, tag: String!): Post  
  deletePost(id: ID!, expectedVersion: Int): Post  
  upvotePost(id: ID!): Post  
  downvotePost(id: ID!): Post  
  updatePost(  
    id: ID!,  
    author: String,  
    title: String,  
    content: String,  
    url: String,  
    expectedVersion: Int!  
  ): Post  
  addPost(  
    author: String!,  
    title: String!,  
    content: String!,  
    url: String!  
  ): Post!
```

```
}
```

- Scegli Save (Salva).
- Nel riquadro Data types (Tipi di dati) a destra, individuare il campo appena creato addComment sul tipo Mutation (Mutazione), quindi scegliere Attach (Collega).
- Nel nome origine dati, scegliere PostDynamoDBTable.
- Incollare il codice seguente nella sezione Configure the request mapping template (Configura il modello di mappatura della richiesta):

```
{
  "version" : "2017-02-28",
  "operation" : "UpdateItem",
  "key" : {
    "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
  },
  "update" : {
    "expression" : "SET comments =
list_append(if_not_exists(comments, :emptyList), :newComment) ADD version :plusOne",
    "expressionValues" : {
      ":emptyList": { "L" : [] },
      ":newComment" : { "L" : [
        { "M": {
          "author": $util.dynamodb.toDynamoDBJson($context.arguments.author),
          "comment": $util.dynamodb.toDynamoDBJson($context.arguments.comment)
        }
      ] },
      ":plusOne" : $util.dynamodb.toDynamoDBJson(1)
    }
  }
}
```

Questa espressione di aggiornamento aggiunge un elenco contenente il nuovo commento all'elenco `comments` esistente. Se l'elenco non esiste già, viene creato.

- Incollare il codice seguente nella sezione Configure the request mapping template (Configura il modello di mappatura della richiesta):

```
$utils.toJson($context.result)
```

- Scegli Save (Salva).

Chiamata dell'API per aggiungere un commento

Ora che hai configurato i resolver, AWS AppSync sa come tradurre le richieste `addComment` in arrivo in operazioni DynamoDB. `UpdateItem`

Proviamo ora ad aggiungere un commento allo stesso post cui abbiamo aggiunto i tag.

- Seleziona la scheda Queries (Query).
- Incollare la query seguente nel riquadro Queries (Query):

```
mutation addComment {
  addComment(
    id:10
    author: "Steve"
    comment: "Such a cute dog."
  ) {
    id
    comments {
      author
      comment
    }
  }
}
```

- Scegliere Execute query (Esegui query) (il pulsante di riproduzione arancione).
- Tutti i post di Nadia verranno visualizzati nel riquadro dei risultati a destra del riquadro della query. La schermata visualizzata dovrebbe risultare simile a quella nell'immagine seguente:

```
{
  "data": {
    "addComment": {
      "id": "10",
      "comments": [
        {
          "author": "Steve",
          "comment": "Such a cute dog."
        }
      ]
    }
  }
}
```

Se esegui la richiesta più volte, all'elenco verranno aggiunti più commenti.

Conclusioni

In questo tutorial, hai creato un'API che ci consente di manipolare gli oggetti Post in DynamoDB AWS AppSync utilizzando GraphQL. Per altre informazioni, vedere la [Informazioni di riferimento sui modelli di mappatura dei resolver](#).

Per eseguire la pulizia, puoi eliminare l'API AppSync GraphQL dalla console.

Per eliminare la tabella DynamoDB e il ruolo IAM che hai creato per questo tutorial, puoi eseguire quanto segue per eliminare lo stack oppure visitare `AWSAppSyncTutorialForAmazonDynamoDB` la console ed eliminare CloudFormation lo stack:

```
aws cloudformation delete-stack \  
  --stack-name AWSAppSyncTutorialForAmazonDynamoDB
```

Utilizzo dei AWS Lambda resolver in AWS AppSync

Note

Ora supportiamo principalmente il runtime `APPSYNC_JS` e la relativa documentazione. [Prendi in considerazione l'utilizzo del runtime `APPSYNC\_JS` e delle relative guide qui.](#)

Puoi usare AWS Lambda with AWS AppSync per risolvere qualsiasi campo GraphQL. Ad esempio, una query GraphQL potrebbe inviare una chiamata a un'istanza Amazon Relational Database Service (Amazon RDS) e una mutazione GraphQL potrebbe scrivere su un flusso Amazon Kinesis. In questa sezione, ti mostreremo come scrivere una funzione Lambda che esegue la logica aziendale basata sull'invocazione di un'operazione sul campo GraphQL.

Creazione di una funzione Lambda

L'esempio seguente mostra una funzione Lambda scritta in `Node.js` che esegue diverse operazioni sui post del blog come parte di un'applicazione per post di blog.

```
exports.handler = (event, context, callback) => {  
  console.log("Received event {}", JSON.stringify(event, 3));  
  var posts = {
```

```

    "1": {"id": "1", "title": "First book", "author": "Author1", "url": "https://amazon.com/", "content": "SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1", "ups": "100", "downs": "10"},
    "2": {"id": "2", "title": "Second book", "author": "Author2", "url": "https://amazon.com", "content": "SAMPLE TEXT AUTHOR 2 SAMPLE TEXT AUTHOR 2 SAMPLE TEXT", "ups": "100", "downs": "10"},
    "3": {"id": "3", "title": "Third book", "author": "Author3", "url": null, "content": null, "ups": null, "downs": null },
    "4": {"id": "4", "title": "Fourth book", "author": "Author4", "url": "https://www.amazon.com/", "content": "SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4", "ups": "1000", "downs": "0"},
    "5": {"id": "5", "title": "Fifth book", "author": "Author5", "url": "https://www.amazon.com/", "content": "SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT", "ups": "50", "downs": "0"} }];

var relatedPosts = {
  "1": [posts['4']],
  "2": [posts['3'], posts['5']],
  "3": [posts['2'], posts['1']],
  "4": [posts['2'], posts['1']],
  "5": []
};

console.log("Got an Invoke Request.");
switch(event.field) {
  case "getPost":
    var id = event.arguments.id;
    callback(null, posts[id]);
    break;
  case "allPosts":
    var values = [];
    for(var d in posts){
      values.push(posts[d]);
    }
    callback(null, values);
    break;
  case "addPost":
    // return the arguments back
    callback(null, event.arguments);
    break;
  case "addPostErrorWithData":
    var id = event.arguments.id;

```

```
        var result = posts[id];
        // attached additional error information to the post
        result.errorMessage = 'Error with the mutation, data has changed';
        result.errorType = 'MUTATION_ERROR';
        callback(null, result);
        break;
    case "relatedPosts":
        var id = event.source.id;
        callback(null, relatedPosts[id]);
        break;
    default:
        callback("Unknown field, unable to resolve" + event.field, null);
        break;
    }
};
```

Questa funzione Lambda recupera un post per ID, aggiunge un post, recupera un elenco di post e recupera i post correlati per un determinato post.

Nota: la funzione Lambda utilizza l'istruzione `on event.field` per determinare quale campo è attualmente in fase di risoluzione.

Crea questa funzione Lambda utilizzando la console di AWS gestione o uno AWS CloudFormation stack. Per creare la funzione da uno CloudFormation stack, puoi usare il seguente comando AWS Command Line Interface (AWS CLI):

```
aws cloudformation create-stack --stack-name AppSyncLambdaExample \
--template-url https://s3.us-west-2.amazonaws.com/awsappsync/resources/lambda/
LambdaCFTemplate.yaml \
--capabilities CAPABILITY_NAMED_IAM
```

Puoi anche avviare lo CloudFormation stack nella AWS regione Stati Uniti occidentali (Oregon) dal tuo AWS account da qui:



Configurare un'origine dati per Lambda

Dopo aver creato la funzione Lambda, accedi all'API GraphQL nella AWS AppSync console, quindi scegli la scheda Data Sources.

Scegli Crea origine dati, inserisci un nome di origine dati descrittivo (ad esempio **Lambda**), quindi per Tipo di origine dati, scegli AWS Lambda funzione. Per Regione, scegli la stessa regione della tua funzione. (Se hai creato la funzione dallo CloudFormation stack fornito, la funzione è probabilmente in US-WEST-2.) Per Function ARN, scegli l'Amazon Resource Name (ARN) della tua funzione Lambda.

Dopo aver scelto la funzione Lambda, puoi creare un nuovo ruolo AWS Identity and Access Management (IAM) (per il quale AWS AppSync assegna le autorizzazioni appropriate) o scegliere un ruolo esistente con la seguente politica in linea:

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "lambda:InvokeFunction"
      ],
      "Resource": "arn:aws:lambda:us-east-1:111122223333:function:LAMBDA_FUNCTION"
    }
  ]
}
```

È inoltre necessario impostare una relazione di fiducia con AWS AppSync per il ruolo IAM nel modo seguente:

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "appsync.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

```
}  
]  
}
```

Creare uno schema GraphQL

Ora che l'origine dati è connessa alla funzione Lambda, crea uno schema GraphQL.

Dall'editor di schemi nella AWS AppSync console, assicurati che lo schema corrisponda allo schema seguente:

```
schema {  
  query: Query  
  mutation: Mutation  
}  
  
type Query {  
  getPost(id:ID!): Post  
  allPosts: [Post]  
}  
  
type Mutation {  
  addPost(id: ID!, author: String!, title: String, content: String, url: String):  
  Post!  
}  
  
type Post {  
  id: ID!  
  author: String!  
  title: String  
  content: String  
  url: String  
  ups: Int  
  downs: Int  
  relatedPosts: [Post]  
}
```

Configura i resolver

Ora che hai registrato un'origine dati Lambda e uno schema GraphQL valido, puoi connettere i campi GraphQL all'origine dati Lambda utilizzando i resolver.

Per creare un resolver, avrai bisogno di modelli di mappatura. Per ulteriori informazioni sui modelli di mappatura, consulta. [Resolver Mapping Template Overview](#)

Per ulteriori informazioni sui modelli di mappatura Lambda, vedere. [Resolver mapping template reference for Lambda](#)

In questo passaggio, si collega un resolver alla funzione Lambda per i seguenti campi: `getPost(id:ID!): Post`, `allPosts: [Post]` e `addPost(id: ID!, author: String!, title: String, content: String, url: String): Post!`
`Post.relatedPosts: [Post]`

Dall'editor di schemi nella AWS AppSync console, sul lato destro, scegli Attach Resolver for. `getPost(id:ID!): Post`

Quindi, nel menu Azione, scegli Update runtime, quindi scegli Unit Resolver (solo VTL).

Successivamente, scegli la tua fonte di dati Lambda. Nella sezione del modello di mappatura della richiesta, scegliere Invoke And Forward Argomenti (Richiama e inoltra argomenti).

Modifica l'oggetto `payload` per aggiungere il nome del campo Il modello avrà un aspetto simile al seguente:

```
{
  "version": "2017-02-28",
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "arguments": $utils.toJson($context.arguments)
  }
}
```

Nella sezione modello di mappatura della risposta, scegliere Return Lambda Result (Restituisci risultato Lambda).

In questo caso, Useremo il modello di base senza apportare modifiche. Avrà un aspetto simile al seguente:

```
$utils.toJson($context.result)
```

Scegli Save (Salva). Il primo resolver è stato ora collegato con successo. Ripeti questa operazione per i campi rimanenti nel modo seguente:

Per il modello di mappatura della richiesta `addPost(id: ID!, author: String!, title: String, content: String, url: String): Post!`:

```
{
  "version": "2017-02-28",
  "operation": "Invoke",
  "payload": {
    "field": "addPost",
    "arguments": $utils.toJson($context.arguments)
  }
}
```

Per il modello di mappatura della risposta `addPost(id: ID!, author: String!, title: String, content: String, url: String): Post!`:

```
$utils.toJson($context.result)
```

Per il modello di mappatura della richiesta `allPosts: [Post]`:

```
{
  "version": "2017-02-28",
  "operation": "Invoke",
  "payload": {
    "field": "allPosts"
  }
}
```

Per il modello di mappatura della risposta `allPosts: [Post]`:

```
$utils.toJson($context.result)
```

Per il modello di mappatura della richiesta `Post.relatedPosts: [Post]`:

```
{
  "version": "2017-02-28",
  "operation": "Invoke",
  "payload": {
    "field": "relatedPosts",
    "source": $utils.toJson($context.source)
  }
}
```



```
}
```

Per il modello di mappatura della risposta `Post.relatedPosts: [Post]`:

```
$utils.toJson($context.result)
```

Testa la tua API GraphQL

Ora che la funzione Lambda è connessa ai resolver GraphQL, puoi eseguire alcune mutazioni e query usando la console o un'applicazione client.

Sul lato sinistro della AWS AppSync console, scegli Queries, quindi incolla il codice seguente:

addPost Mutation

```
mutation addPost {  
  addPost(  
    id: 6  
    author: "Author6"  
    title: "Sixth book"  
    url: "https://www.amazon.com/"  
    content: "This is the book is a tutorial for using GraphQL with AWS AppSync."  
  ) {  
    id  
    author  
    title  
    content  
    url  
    ups  
    downs  
  }  
}
```

getPost Query

```
query getPost {  
  getPost(id: "2") {  
    id  
    author  
    title  
  }  
}
```

```
        content
        url
        ups
        downs
    }
}
```

allPosts Query

```
query allPosts {
  allPosts {
    id
    author
    title
    content
    url
    ups
    downs
    relatedPosts {
      id
      title
    }
  }
}
```

Errori di restituzione

Qualsiasi risoluzione di campo può causare un errore. Con AWS AppSync, puoi generare errori dalle seguenti fonti:

- Modello di mappatura della richiesta o della risposta
- funzione Lambda

Dal modello di mappatura

Per generare errori intenzionali, è possibile utilizzare il metodo `$utils.error` helper del modello Velocity Template Language (VTL). L'argomento può essere un valore `errorMessage`, un valore `errorType` e un valore facoltativo `data`. Il valore `data` è utile per restituire dati aggiuntivi al client quando è stato generato un errore. L'oggetto `data` verrà aggiunto a `errors` nella risposta finale di GraphQL.

L'esempio seguente mostra come usarlo nel modello di mappatura della risposta

`Post.relatedPosts: [Post]:`

```
$utils.error("Failed to fetch relatedPosts", "LambdaFailure", $context.result)
```

Ciò restituirà una risposta di GraphQL simile alla seguente:

```
{
  "data": {
    "allPosts": [
      {
        "id": "2",
        "title": "Second book",
        "relatedPosts": null
      },
      ...
    ]
  },
  "errors": [
    {
      "path": [
        "allPosts",
        0,
        "relatedPosts"
      ],
      "errorType": "LambdaFailure",
      "locations": [
        {
          "line": 5,
          "column": 5
        }
      ],
      "message": "Failed to fetch relatedPosts",
      "data": [
        {
          "id": "2",
          "title": "Second book"
        },
        {
          "id": "1",
          "title": "First book"
        }
      ]
    }
  ]
}
```

```
    }
  ]
}
```

Dove `allPosts[0].relatedPosts` è null a causa dell'errore e `errorMessage`, `errorType` e `data` sono presenti nell'oggetto `data.errors[0]`.

Dalla funzione Lambda

AWS AppSync comprende anche gli errori generati dalla funzione Lambda. Il modello di programmazione Lambda consente di generare errori gestiti. Se la funzione Lambda genera un errore, AWS AppSync non riesce a risolvere il campo corrente. Nella risposta viene impostato solo il messaggio di errore restituito da Lambda. Attualmente, non è possibile restituire dati estranei al client generando un errore dalla funzione Lambda.

Nota: se la funzione Lambda genera un errore non gestito, AWS AppSync utilizza il messaggio di errore impostato da Lambda.

La funzione Lambda seguente genera un errore:

```
exports.handler = (event, context, callback) => {
  console.log("Received event {}", JSON.stringify(event, 3));
  callback("I fail. Always.");
};
```

Ciò restituirà una risposta di GraphQL simile alla seguente:

```
{
  "data": {
    "allPosts": [
      {
        "id": "2",
        "title": "Second book",
        "relatedPosts": null
      },
      ...
    ]
  },
  "errors": [
    {
      "path": [
```

```

        "allPosts",
        0,
        "relatedPosts"
    ],
    "errorType": "Lambda:Handled",
    "locations": [
        {
            "line": 5,
            "column": 5
        }
    ],
    "message": "I fail. Always."
}
]
}

```

Caso d'uso avanzato: Batching

La funzione Lambda in questo esempio ha un `relatedPosts` campo che restituisce un elenco di post correlati per un determinato post. Nelle query di esempio, l'invocazione del `allPosts` campo dalla funzione Lambda restituisce cinque post. Poiché abbiamo specificato che vogliamo risolvere anche `relatedPosts` per ogni post restituito, l'operazione `relatedPosts` sul campo viene richiamata cinque volte.

```

query allPosts {
  allPosts { // 1 Lambda invocation - yields 5 Posts
    id
    author
    title
    content
    url
    ups
    downs
    relatedPosts { // 5 Lambda invocations - each yields 5 posts
      id
      title
    }
  }
}

```

Anche se questo potrebbe non sembrare importante in questo esempio specifico, questo eccesso di recupero aggravato può compromettere rapidamente l'applicazione.

Se, ad esempio, dovessimo recuperare di nuovo `relatedPosts` sui `Posts` correlati restituiti nella stessa query, il numero di chiamate aumenterebbe notevolmente.

```

query allPosts {
  allPosts {    // 1 Lambda invocation - yields 5 Posts
    id
    author
    title
    content
    url
    ups
    downs
    relatedPosts {    // 5 Lambda invocations - each yield 5 posts = 5 x 5 Posts
      id
      title
      relatedPosts {    // 5 x 5 Lambda invocations - each yield 5 posts = 25 x 5
        id
        title
        author
      }
    }
  }
}

```

Posts

In questa query relativamente semplice, AWS AppSync richiamerebbe la funzione Lambda $1 + 5 + 25 = 31$ volte.

Si tratta di una sfida piuttosto comune a cui si fa spesso riferimento come problema $N+1$, (nel nostro caso, $N = 5$) che può determinare un aumento della latenza e dei costi dell'applicazione.

Un approccio alla soluzione di questo problema consiste nel raggruppare in batch richieste del resolver di campo simili. In questo esempio, invece di fare in modo che la funzione Lambda risolva un elenco di post correlati per un singolo post, potrebbe invece risolvere un elenco di post correlati per un determinato batch di post.

Per provare questa funzionalità, modifichiamo il resolver `Post.relatedPosts: [Post]` in un resolver abilitato per i batch.

Sul lato destro della AWS AppSync console, scegli il `Post.relatedPosts: [Post]` resolver esistente. Modifica il modello di mappatura della richiesta nel modo seguente:

```
{
  "version": "2017-02-28",
  "operation": "BatchInvoke",
  "payload": {
    "field": "relatedPosts",
    "source": $utils.toJson($context.source)
  }
}
```

Solo il campo `operation` è stato modificato da `Invoke` a `BatchInvoke`. Il campo `payload` ora diventa un array di tutto ciò che è specificato nel modello. In questo esempio, la funzione Lambda riceve quanto segue come input:

```
[
  {
    "field": "relatedPosts",
    "source": {
      "id": 1
    }
  },
  {
    "field": "relatedPosts",
    "source": {
      "id": 2
    }
  },
  ...
]
```

Quando `BatchInvoke` è specificata nel modello di mappatura delle richieste, la funzione Lambda riceve un elenco di richieste e restituisce un elenco di risultati.

In particolare, l'elenco dei risultati deve corrispondere alla dimensione e all'ordine delle voci del payload della richiesta in modo che AWS AppSync possa corrispondere ai risultati di conseguenza.

In questo esempio di batch, la funzione Lambda restituisce un batch di risultati come segue:

```
[
  [{"id":"2","title":"Second book"}, {"id":"3","title":"Third book"}], //
  relatedPosts for id=1
  [{"id":"3","title":"Third book"}]
  // relatedPosts for id=2
]
```

]

La seguente funzione Lambda in Node.js dimostra questa funzionalità di batch per il `Post.relatedPosts` campo come segue:

```
exports.handler = (event, context, callback) => {
  console.log("Received event {}", JSON.stringify(event, 3));
  var posts = {
    "1": {"id": "1", "title": "First book", "author": "Author1", "url": "https://amazon.com/", "content": "SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1", "ups": "100", "downs": "10"},
    "2": {"id": "2", "title": "Second book", "author": "Author2", "url": "https://amazon.com", "content": "SAMPLE TEXT AUTHOR 2 SAMPLE TEXT AUTHOR 2 SAMPLE TEXT", "ups": "100", "downs": "10"},
    "3": {"id": "3", "title": "Third book", "author": "Author3", "url": null, "content": null, "ups": null, "downs": null },
    "4": {"id": "4", "title": "Fourth book", "author": "Author4", "url": "https://www.amazon.com/", "content": "SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4", "ups": "1000", "downs": "0"},
    "5": {"id": "5", "title": "Fifth book", "author": "Author5", "url": "https://www.amazon.com/", "content": "SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT", "ups": "50", "downs": "0"} };

  var relatedPosts = {
    "1": [posts['4']],
    "2": [posts['3'], posts['5']],
    "3": [posts['2'], posts['1']],
    "4": [posts['2'], posts['1']],
    "5": []
  };

  console.log("Got a BatchInvoke Request. The payload has %d items to resolve.", event.length);
  // event is now an array
  var field = event[0].field;
  switch(field) {
    case "relatedPosts":
      var results = [];
      // the response MUST contain the same number
      // of entries as the payload array
      for (var i=0; i< event.length; i++) {
```



```

        console.log("post {}", JSON.stringify(event[i].source));
        results.push(relatedPosts[event[i].source.id]);
    }
    console.log("results {}", JSON.stringify(results));
    callback(null, results);
    break;
default:
    callback("Unknown field, unable to resolve" + field, null);
    break;
}
};

```

Restituzione di errori individuali

Gli esempi precedenti mostrano che è possibile restituire un singolo errore dalla funzione Lambda o generare un errore dai modelli di mappatura. Per le chiamate in batch, la generazione di un errore dalla funzione Lambda contrassegna un intero batch come fallito. Questo potrebbe essere accettabile per scenari specifici in cui si verifica un errore irreversibile, ad esempio una connessione non riuscita a un data store. Tuttavia, nei casi in cui alcuni elementi del batch abbiano esito positivo e altri falliscano, è possibile restituire sia errori che dati validi. Poiché AWS AppSync richiede la risposta in batch agli elementi dell'elenco che corrispondono alla dimensione originale del batch, è necessario definire una struttura di dati in grado di differenziare i dati validi da un errore.

Ad esempio, se si prevede che la funzione Lambda restituisca un batch di post correlati, è possibile scegliere di restituire un elenco di *Response* oggetti in cui ogni oggetto contiene dati opzionali, campi *ErrorMessage* ed *ErrorType*. La presenza del campo *errorMessage* indica un errore.

Il codice seguente mostra come aggiornare la funzione Lambda:

```

exports.handler = (event, context, callback) => {
    console.log("Received event {}", JSON.stringify(event, 3));
    var posts = {
        "1": {"id": "1", "title": "First book", "author": "Author1", "url": "https://amazon.com/", "content": "SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1", "ups": "100", "downs": "10"},
        "2": {"id": "2", "title": "Second book", "author": "Author2", "url": "https://amazon.com", "content": "SAMPLE TEXT AUTHOR 2 SAMPLE TEXT AUTHOR 2 SAMPLE TEXT", "ups": "100", "downs": "10"},
        "3": {"id": "3", "title": "Third book", "author": "Author3", "url": null, "content": null, "ups": null, "downs": null },
    }
}

```

```

    "4": {"id": "4", "title": "Fourth book", "author": "Author4", "url": "https://
www.amazon.com/", "content": "SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
AUTHOR 4 SAMPLE TEXT AUTHOR 4", "ups": "1000", "downs": "0"},
    "5": {"id": "5", "title": "Fifth book", "author": "Author5", "url": "https://
www.amazon.com/", "content": "SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT
AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT", "ups": "50", "downs": "0"} }];

var relatedPosts = {
  "1": [posts['4']],
  "2": [posts['3'], posts['5']],
  "3": [posts['2'], posts['1']],
  "4": [posts['2'], posts['1']],
  "5": []
};

console.log("Got a BatchInvoke Request. The payload has %d items to resolve.",
event.length);
// event is now an array
var field = event[0].field;
switch(field) {
  case "relatedPosts":
    var results = [];
    results.push({ 'data': relatedPosts['1'] });
    results.push({ 'data': relatedPosts['2'] });
    results.push({ 'data': null, 'errorMessage': 'Error Happened', 'errorType':
'ERROR' });
    results.push(null);
    results.push({ 'data': relatedPosts['3'], 'errorMessage': 'Error Happened
with last result', 'errorType': 'ERROR' });
    callback(null, results);
    break;
  default:
    callback("Unknown field, unable to resolve" + field, null);
    break;
}
};

```

Per questo esempio, il seguente modello di mappatura delle risposte analizza ogni elemento della funzione Lambda e genera gli eventuali errori che si verificano:

```
#if( $context.result && $context.result.errorMessage )
```

```
$utils.error($context.result.errorMessage, $context.result.errorType,  
$context.result.data)  
#else  
$utils.toJson($context.result.data)  
#end
```

Questo esempio restituirà una risposta di GraphQL simile alla seguente:

```
{  
  "data": {  
    "allPosts": [  
      {  
        "id": "1",  
        "relatedPostsPartialErrors": [  
          {  
            "id": "4",  
            "title": "Fourth book"  
          }  
        ]  
      },  
      {  
        "id": "2",  
        "relatedPostsPartialErrors": [  
          {  
            "id": "3",  
            "title": "Third book"  
          },  
          {  
            "id": "5",  
            "title": "Fifth book"  
          }  
        ]  
      },  
      {  
        "id": "3",  
        "relatedPostsPartialErrors": null  
      },  
      {  
        "id": "4",  
        "relatedPostsPartialErrors": null  
      },  
      {  
        "id": "5",
```

```
      "relatedPostsPartialErrors": null
    }
  ]
},
"errors": [
  {
    "path": [
      "allPosts",
      2,
      "relatedPostsPartialErrors"
    ],
    "errorType": "ERROR",
    "locations": [
      {
        "line": 4,
        "column": 9
      }
    ],
    "message": "Error Happened"
  },
  {
    "path": [
      "allPosts",
      4,
      "relatedPostsPartialErrors"
    ],
    "data": [
      {
        "id": "2",
        "title": "Second book"
      },
      {
        "id": "1",
        "title": "First book"
      }
    ],
    "errorType": "ERROR",
    "locations": [
      {
        "line": 4,
        "column": 9
      }
    ],
    "message": "Error Happened with last result"
  }
]
```

```
}  
]  
}
```

Configurazione della dimensione massima di batch

Per impostazione predefinita, quando si utilizza `BatchInvoke`, AWS AppSync invia richieste alla funzione Lambda in batch composti da un massimo di cinque elementi. Puoi configurare la dimensione massima del batch dei tuoi resolver Lambda.

Per configurare la dimensione massima di batch su un resolver, usa il seguente comando in `()`: AWS Command Line Interface AWS CLI

```
$ aws appsync create-resolver --api-id <api-id> --type-name Query --field-name  
relatedPosts \  
--request-mapping-template "<template>" --response-mapping-template "<template>" --  
data-source-name "<lambda-datasource>" \  
--max-batch-size X
```

Note

Quando si fornisce un modello di mappatura delle richieste, è necessario utilizzare l'operazione `BatchInvoke` per utilizzare il batch.

È inoltre possibile utilizzare il seguente comando per abilitare e configurare il batching su Direct Lambda Resolver:

```
$ aws appsync create-resolver --api-id <api-id> --type-name Query --field-name  
relatedPosts \  
--data-source-name "<lambda-datasource>" \  
--max-batch-size X
```

Configurazione della dimensione massima di batch con modelli VTL

Per i Lambda Resolver che dispongono di modelli VTL su richiesta, la dimensione massima del batch non avrà alcun effetto a meno che non l'abbiano specificata direttamente come operazione in VTL. Allo stesso modo, se si esegue una mutazione di primo livello, il batching non viene eseguito per le mutazioni perché la specifica GraphQL richiede che le mutazioni parallele vengano eseguite in sequenza.

Ad esempio, prendiamo le seguenti mutazioni:

```
type Mutation {  
  putItem(input: Item): Item  
  putItems(inputs: [Item]): [Item]  
}
```

Usando la prima mutazione, possiamo crearne 10 Items come mostrato nel frammento seguente:

```
mutation MyMutation {  
  v1: putItem($someItem1) {  
    id,  
    name  
  }  
  v2: putItem($someItem2) {  
    id,  
    name  
  }  
  v3: putItem($someItem3) {  
    id,  
    name  
  }  
  v4: putItem($someItem4) {  
    id,  
    name  
  }  
  v5: putItem($someItem5) {  
    id,  
    name  
  }  
  v6: putItem($someItem6) {  
    id,  
    name  
  }  
  v7: putItem($someItem7) {  
    id,  
    name  
  }  
  v8: putItem($someItem8) {  
    id,  
    name  
  }  
  v9: putItem($someItem9) {
```

```
      id,  
      name  
    }  
    v10: putItem($someItem10) {  
      id,  
      name  
    }  
  }  
}
```

In questo esempio, non Items verranno raggruppati in un gruppo di 10 anche se la dimensione massima del batch è impostata su 10 nel Lambda Resolver. Verranno invece eseguiti in sequenza secondo le specifiche GraphQL.

Per eseguire una vera mutazione in batch, puoi seguire l'esempio seguente usando la seconda mutazione:

```
mutation MyMutation {  
  putItems([$someItem1, $someItem2, $someItem3,$someItem4, $someItem5, $someItem6,  
    $someItem7, $someItem8, $someItem9, $someItem10]) {  
    id,  
    name  
  }  
}
```

Per ulteriori informazioni sull'utilizzo del batching con Direct Lambda Resolver, vedere. [Resolver Lambda diretti](#)

Utilizzo dei OpenSearch resolver Amazon Service in AWS AppSync

Note

Ora supportiamo principalmente il runtime APPSYNC_JS e la relativa documentazione. [Prendi in considerazione l'utilizzo del runtime APPSYNC_JS e delle relative guide qui.](#)

AWS AppSync supporta l'utilizzo di Amazon OpenSearch Service da domini che hai fornito nel tuo AWS account, a condizione che non esistano all'interno di un VPC. Dopo che sono stati assegnati i domini, è possibile connettersi a essi tramite un'origine dati, a quel punto è possibile configurare un resolver nello schema per eseguire operazioni di GraphQL, come ad esempio query, mutazioni e iscrizioni. Questo tutorial fornirà una descrizione di alcuni esempi comuni.

Per ulteriori informazioni, consulta il [Resolver Mapping Template Reference](#) per. OpenSearch

Impostazione One-Click

Per configurare automaticamente un endpoint GraphQL con AWS AppSync Amazon OpenSearch Service configurato, puoi utilizzare questo modello: AWS CloudFormation

A yellow button with a blue play icon and the text "Launch Stack".

Una volta completata la AWS CloudFormation distribuzione, puoi passare direttamente all'esecuzione di query e mutazioni [GraphQL](#).

Crea un nuovo dominio di servizio OpenSearch

Per iniziare con questo tutorial, è necessario un dominio di OpenSearch servizio esistente. Se non si dispone di un dominio, è possibile usare il campione seguente. Tieni presente che possono essere necessari fino a 15 minuti per la creazione di un dominio di OpenSearch servizio prima di poter passare all'integrazione con una fonte di AWS AppSync dati.

```
aws cloudformation create-stack --stack-name AppSyncOpenSearch \  
--template-url https://s3.us-west-2.amazonaws.com/awsappsync/resources/elasticsearch/  
ESResolverCFTemplate.yaml \  
--parameters ParameterKey=OSDomainName,ParameterValue=ddtestdomain  
ParameterKey=Tier,ParameterValue=development \  
--capabilities CAPABILITY_NAMED_IAM
```

Puoi avviare il seguente AWS CloudFormation stack nella regione US West 2 (Oregon) nel tuo account: AWS

A yellow button with a blue play icon and the text "Launch Stack".

Configura l'origine dei dati per il servizio OpenSearch

Dopo aver creato il dominio OpenSearch Service, accedi all'API AWS AppSync GraphQL e scegli la scheda Data Sources. Scegli Nuovo e inserisci un nome descrittivo per l'origine dati, ad esempio «oss». Quindi scegli il OpenSearch dominio Amazon per il tipo di origine dati, scegli la regione appropriata e dovresti vedere il tuo dominio OpenSearch di servizio elencato. Dopo averlo

selezionato, puoi creare un nuovo ruolo e AWS AppSync assegnare le autorizzazioni appropriate al ruolo oppure puoi scegliere un ruolo esistente, che abbia la seguente politica in linea:

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Stmt1234234",
      "Effect": "Allow",
      "Action": [
        "es:ESHttpDelete",
        "es:ESHttpHead",
        "es:ESHttpGet",
        "es:ESHttpPost",
        "es:ESHttpPut"
      ],
      "Resource": [
        "arn:aws:es:us-east-1:111122223333:domain/democluster/*"
      ]
    }
  ]
}
```

Dovrai anche stabilire una relazione di fiducia con per quel ruolo: AWS AppSync

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "appsync.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

```
}
```

Inoltre, il dominio OpenSearch Service ha una propria politica di accesso che puoi modificare tramite la console di Amazon OpenSearch Service. Dovrai aggiungere una politica simile alla seguente, con le azioni e le risorse appropriate per il dominio del OpenSearch servizio. Tieni presente che il ruolo Principal sarà il ruolo dell'origine AppSync dati, che se lasci che sia la console a crearlo, può essere trovato nella console IAM.

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::111122223333:role/service-role/
APPSYNC_DATASOURCE_ROLE"
      },
      "Action": [
        "es:ESHttpDelete",
        "es:ESHttpHead",
        "es:ESHttpGet",
        "es:ESHttpPost",
        "es:ESHttpPut"
      ],
      "Resource": "arn:aws:es:us-east-1:111122223333:domain/DOMAIN_NAME/*"
    }
  ]
}
```

Collegamento di un Resolver

Ora che l'origine dati è connessa al tuo dominio OpenSearch Service, puoi connetterla allo schema GraphQL con un resolver, come mostrato nell'esempio seguente:

```
schema {
  query: Query
  mutation: Mutation
}
```

```

}

type Query {
  getPost(id: ID!): Post
  allPosts: [Post]
}

type Mutation {
  addPost(id: ID!, author: String, title: String, url: String, ups: Int, downs: Int,
content: String): AWSJSON
}

type Post {
  id: ID!
  author: String
  title: String
  url: String
  ups: Int
  downs: Int
  content: String
}
...

```

Si noti che vi è un tipo `Post` definito dall'utente con un campo di `id`. Negli esempi seguenti, supponiamo che esista un processo (che può essere automatizzato) per inserire questo tipo nel dominio del OpenSearch servizio, che verrebbe mappato alla radice del percorso di `/post/_doc`, where `post` è l'indice. Da questo percorso principale, è possibile eseguire ricerche su singoli documenti, ricerche con caratteri jolly o ricerche su più documenti con `/id/post*` un percorso di `/post/_search`. Ad esempio, se avete un altro tipo chiamato **User**, potete indicizzare i documenti in base a un nuovo indice chiamato **user**, quindi eseguire ricerche con un percorso di `/user/_search`.

Dall'editor dello schema nella AWS AppSync console, modifica lo `Posts` schema precedente per includere una `searchPosts` query:

```

type Query {
  getPost(id: ID!): Post
  allPosts: [Post]
  searchPosts: [Post]
}

```

Salvare lo schema. Sul lato destro, per `searchPosts`, scegliere `Allega resolver`. Nel menu `Azione`, scegli `Update runtime`, quindi scegli `Unit Resolver` (solo VTL). Quindi, scegli la fonte dei dati del `OpenSearch` servizio. Nella sezione modello di mappatura della richiesta, selezionare il menu a discesa per `Post` della query per ottenere un modello di base. Modificare il path in `/post/_search`. Avrà un aspetto simile al seguente:

```
{
  "version": "2017-02-28",
  "operation": "GET",
  "path": "/post/_search",
  "params": {
    "headers": {},
    "queryString": {},
    "body": {
      "from": 0,
      "size": 50
    }
  }
}
```

Ciò presuppone che lo schema precedente contenga documenti che sono stati indicizzati in `OpenSearch Service` sotto il campo `post`. Se si strutturano i dati in modo diverso, è necessario aggiornarli di conseguenza.

Nella sezione del modello di mappatura delle risposte, è necessario specificare il `_source` filtro appropriato se si desidera recuperare i risultati dei dati da una query di `OpenSearch` servizio e tradurli in GraphQL. Usare il modello seguente:

```
[
  #foreach($entry in $context.result.hits.hits)
  #if( $velocityCount > 1 ) , #end
  $utils.toJson($entry.get("_source"))
  #end
]
```

Modificare le ricerche

Il modello di mappatura della richiesta precedente esegue una semplice query per tutti i record. Se si desidera eseguire la ricerca per un autore specifico e se si desidera, inoltre, che l'autore sia un

argomento definito nella query di GraphQL, Nell'editor dello schema della console di AWS AppSync aggiungere una query `allPostsByAuthor`:

```
type Query {  
  getPost(id: ID!): Post  
  allPosts: [Post]  
  allPostsByAuthor(author: String!): [Post]  
  searchPosts: [Post]  
}
```

Ora scegli Attach resolver e seleziona l'origine dati del OpenSearch servizio, ma usa il seguente esempio nel modello di mappatura delle risposte:

```
{  
  "version": "2017-02-28",  
  "operation": "GET",  
  "path": "/post/_search",  
  "params": {  
    "headers": {},  
    "queryString": {},  
    "body": {  
      "from": 0,  
      "size": 50,  
      "query": {  
        "match": {  
          "author": $util.toJson($context.arguments.author)  
        }  
      }  
    }  
  }  
}
```

Si noti che body viene popolato con una query del termine per il campo `author`, che è già passato attraverso il client come argomento. È possibile facoltativamente avere informazioni preinserite, come ad esempio il testo standard, o anche l'uso di altre [utilità](#).

Se si usa questo resolver, compilare il modello di mappatura della risposta con le stesse informazioni dell'esempio precedente.

Aggiungere dati al servizio OpenSearch

Potresti voler aggiungere dati al tuo dominio di OpenSearch servizio come risultato di una mutazione GraphQL. Si tratta di un meccanismo potente per le ricerche e altri scopi. Poiché è possibile utilizzare gli abbonamenti GraphQL per [rendere i dati in tempo reale](#), funge da meccanismo per notificare ai clienti gli aggiornamenti dei dati nel dominio di servizio. OpenSearch

Torna alla pagina Schema nella AWS AppSync console e seleziona Attach resolver per la mutazione. `addPost()` Seleziona nuovamente l'origine dati del OpenSearch servizio e utilizza il seguente modello di mappatura delle risposte per lo schema: `Posts`

```
{
  "version": "2017-02-28",
  "operation": "PUT",
  "path": $util.toJson("/post/_doc/$context.arguments.id"),
  "params": {
    "headers": {},
    "queryString": {},
    "body": {
      "id": $util.toJson($context.arguments.id),
      "author": $util.toJson($context.arguments.author),
      "ups": $util.toJson($context.arguments.ups),
      "downs": $util.toJson($context.arguments.downs),
      "url": $util.toJson($context.arguments.url),
      "content": $util.toJson($context.arguments.content),
      "title": $util.toJson($context.arguments.title)
    }
  }
}
```

Così come illustrato in precedenza, questo è un esempio di come i dati potrebbero essere strutturati. Se si dispone di nomi di campi o di indici diversi, è necessario aggiornare il path e il body in base alle esigenze. In questo esempio viene inoltre illustrato come usare `$context.arguments` per popolare il modello dagli argomenti della mutazione di GraphQL.

Prima di procedere, utilizzate il seguente modello di mappatura delle risposte, che restituirà come output il risultato dell'operazione di mutazione o le informazioni sull'errore:

```
#if($context.error)
  $util.toJson($ctx.error)
#else
```

```
$util.toJson($context.result)
#end
```

Recupero di un documento individuale

Infine, se si desidera usare la query `getPost(id:ID)` nello schema per restituire un documento singolo, trovare questa query nell'editor dello schema della console di AWS AppSync e selezionare **Allega resolver**. Seleziona nuovamente l'origine dati del OpenSearch servizio e utilizza il seguente modello di mappatura:

```
{
  "version": "2017-02-28",
  "operation": "GET",
  "path": $util.toJson("post/_doc/$context.arguments.id"),
  "params": {
    "headers": {},
    "queryString": {},
    "body": {}
  }
}
```

Poiché il path sopra indicato usa l'argomento `id` con un corpo vuoto, questo restituisce il singolo documento. Tuttavia, è necessario usare il seguente modello di mappatura della risposta, perché ora si restituisce un elemento singolo e non un elenco:

```
$utils.toJson($context.result.get("_source"))
```

Eseguire query e mutazioni

Ora dovresti essere in grado di eseguire operazioni GraphQL sul tuo dominio di OpenSearch servizio. Vai alla scheda Query della AWS AppSync console e aggiungi un nuovo record:

```
mutation addPost {
  addPost (
    id: "12345"
    author: "Fred"
    title: "My first book"
    content: "This will be fun to write!"
    url: "publisher website",
    ups: 100,
```

```
        downs: 20
      )
    }
  }
```

Vedrai il risultato della mutazione sulla destra. Allo stesso modo, ora puoi eseguire una `searchPosts` query sul tuo dominio OpenSearch di servizio:

```
query searchPosts {
  searchPosts {
    id
    title
    author
    content
  }
}
```

Best practice

- OpenSearch Il servizio deve essere utilizzato per l'interrogazione dei dati, non come database principale. [Potresti voler utilizzare OpenSearch Service insieme ad Amazon DynamoDB, come indicato in Combined GraphQL Resolvers.](#)
- Concedi l'accesso al tuo dominio solo permettendo al ruolo di servizio di accedere al cluster. AWS AppSync
- È possibile iniziare con una soluzione di base in fase di sviluppo, con il cluster dal prezzo più basso, e quindi spostarsi in un cluster più grande con elevata disponibilità quando si passa alla produzione.

Utilizzo di resolver locali in AWS AppSync

Note

Ora supportiamo principalmente il runtime APPSYNC_JS e la relativa documentazione. [Prendi in considerazione l'utilizzo del runtime APPSYNC_JS e delle relative guide qui.](#)

AWS AppSync consente di utilizzare fonti di dati supportate (AWS Lambda Amazon DynamoDB o OpenSearch Amazon Service) per eseguire varie operazioni. In alcuni scenari, tuttavia, una chiamata a un'origine dati supportata può non essere necessaria.

È in queste situazioni che il resolver locale risulta utile. Invece di chiamare un'origine dati remota, il resolver locale inoltra semplicemente il risultato del modello di mappatura della richiesta al modello di mappatura della risposta. La risoluzione del campo rimarrà in AWS AppSync.

I resolver locali sono utili per diversi casi d'uso. Quello più comune è la pubblicazione di notifiche senza attivare una chiamata a un'origine dati. Per dimostrare questo caso d'uso creeremo un'applicazione di paging in cui gli utenti possono inviarsi messaggi. In questo esempio vengono usate le sottoscrizioni. Quindi, se non hai familiarità con le sottoscrizioni, puoi seguire il tutorial [Dati in tempo reale](#).

Creare l'applicazione di paging

In questa applicazione di paging, i client possono iscriversi a una casella di posta e inviare pagine ad altri client. Ogni pagina include un messaggio. Lo schema è il seguente:

```
schema {
  query: Query
  mutation: Mutation
  subscription: Subscription
}

type Subscription {
  inbox(to: String!): Page
  @aws_subscribe(mutations: ["page"])
}

type Mutation {
  page(body: String!, to: String!): Page!
}

type Page {
  from: String
  to: String!
  body: String!
  sentAt: String!
}

type Query {
  me: String
}
```

Collegiamo ora un resolver al campo `Mutation.page`. Nel riquadro Schema, fai clic su **Attach Resolver** (Collega resolver) accanto alla definizione del campo nel riquadro destro. Crea una nuova fonte di dati di tipo **Nessuno** e assegna un nome. `PageDataSource`

Per il modello di mappatura della richiesta immetti:

```
{
  "version": "2017-02-28",
  "payload": {
    "body": $util.toJson($context.arguments.body),
    "from": $util.toJson($context.identity.username),
    "to": $util.toJson($context.arguments.to),
    "sentAt": "$util.time.nowISO8601()"
  }
}
```

E per il modello di mappatura della risposta, seleziona l'impostazione predefinita **Forward the result** (Inoltra il risultato). Salva il resolver. L'applicazione è pronta per iniziare a inviare messaggi.

Inviare e iscriversi alle pagine

Perché i client possano ricevere le pagine, devono prima iscriversi a una casella di posta.

Nel riquadro **Queries (Query)** eseguiamo la sottoscrizione `inbox`:

```
subscription Inbox {
  inbox(to: "Nadia") {
    body
    to
    from
    sentAt
  }
}
```

Nadia riceverà le pagine ogni volta che la mutazione `Mutation.page` viene richiamata. La mutazione viene richiamata tramite la relativa esecuzione:

```
mutation Page {
  page(to: "Nadia", body: "Hello, World!") {
    body
  }
}
```

```
    to
    from
    sentAt
  }
}
```

Abbiamo appena dimostrato l'uso dei resolver locali, inviando una pagina e ricevendola senza uscire da AWS AppSync.

Combinazione di resolver GraphQL in AWS AppSync

Note

Ora supportiamo principalmente il runtime APPSYNC_JS e la relativa documentazione. [Prendi in considerazione l'utilizzo del runtime APPSYNC_JS e delle relative guide qui.](#)

I resolver e i campi in uno schema di GraphQL hanno un rapporto 1:1 con un elevato grado di flessibilità. Poiché un'origine dati è configurata su un resolver indipendentemente dallo schema, c'è la possibilità di risolvere o manipolare i tipi di GraphQL tramite diverse origini dati, mescolandole o combinandole in uno schema per soddisfare al meglio le esigenze.

Gli scenari di esempio seguenti mostrano come combinare e abbinare le fonti di dati nello schema. Prima di iniziare, ti consigliamo di acquisire dimestichezza con la configurazione di sorgenti di dati e resolver per Amazon AWS Lambda DynamoDB e Amazon OpenSearch Service, come descritto nei tutorial precedenti.

Schema di esempio

Lo schema seguente ha un tipo Post con 3 Query operazioni e 3 Mutation operazioni definite:

```
type Post {
  id: ID!
  author: String!
  title: String
  content: String
  url: String
  ups: Int
  downs: Int
}
```

```
    version: Int!
  }

  type Query {
    allPost: [Post]
    getPost(id: ID!): Post
    searchPosts: [Post]
  }

  type Mutation {
    addPost(
      id: ID!,
      author: String!,
      title: String,
      content: String,
      url: String
    ): Post
    updatePost(
      id: ID!,
      author: String!,
      title: String,
      content: String,
      url: String,
      ups: Int!,
      downs: Int!,
      expectedVersion: Int!
    ): Post
    deletePost(id: ID!): Post
  }
```

In questo esempio si dispone di un totale di 6 resolver da allegare. Un modo possibile sarebbe quello di far sì che tutti questi dati provengano da una tabella Amazon DynamoDB, `Posts` chiamata, `AllPosts` dove esegue una scansione `searchPosts` ed esegue una query, come indicato nel [DynamoDB Resolver Mapping Template Reference](#). Tuttavia, esistono alternative per soddisfare le esigenze aziendali, come la risoluzione di queste query GraphQL da Lambda o Service. OpenSearch

Modificare i dati tramite resolver

Potrebbe essere necessario restituire i risultati di un database come DynamoDB (o Amazon Aurora) ai client con alcuni attributi modificati. Questo potrebbe essere dovuto alla formattazione dei tipi di dati, ad esempio differenze di timestamp sui client, oppure gestire problemi di retrocompatibilità. A scopo illustrativo, nell'esempio seguente, una AWS Lambda funzione manipola i voti positivi e

negativi per i post del blog assegnando loro numeri casuali ogni volta che viene richiamato il resolver GraphQL:

```
'use strict';
const doc = require('dynamodb-doc');
const dynamo = new doc.DynamoDB();

exports.handler = (event, context, callback) => {
  const payload = {
    TableName: 'Posts',
    Limit: 50,
    Select: 'ALL_ATTRIBUTES',
  };

  dynamo.scan(payload, (err, data) => {
    const result = { data: data.Items.map(item =>{
      item.ups = parseInt(Math.random() * (50 - 10) + 10, 10);
      item.downs = parseInt(Math.random() * (20 - 0) + 0, 10);
      return item;
    }) };
    callback(err, result.data);
  });
};
```

Questa è una funzione di Lambda perfettamente valida e potrebbe essere associata al campo `AllPosts` nello schema di GraphQL in modo che qualsiasi query che restituisce tutti i risultati ottenga numeri casuali per i voti in alto/in basso.

DynamoDB e Service OpenSearch

Per alcune applicazioni, è possibile eseguire mutazioni o semplici query di ricerca su DynamoDB e disporre di un processo in background per trasferire i documenti al Service. OpenSearch. È quindi possibile collegare semplicemente il `searchPosts` Resolver all'origine dati del OpenSearch servizio e restituire i risultati della ricerca (dai dati originati in DynamoDB) utilizzando una query GraphQL. Questo può essere estremamente potente quando si aggiungono operazioni di ricerca avanzate per applicazioni come parole chiave, combinazioni ambigue di parole o anche ricerche geospaziali. Il trasferimento di dati da DynamoDB può essere effettuato tramite un processo ETL o, in alternativa, è possibile eseguire lo streaming da DynamoDB utilizzando Lambda. Puoi lanciare un esempio completo di ciò utilizzando il seguente AWS CloudFormation stack nella regione US West 2 (Oregon) nel tuo account: AWS

Launch Stack

Lo schema in questo esempio consente di aggiungere post utilizzando un resolver DynamoDB come segue:

```
mutation add {  
  putPost(author:"Nadia"  
    title:"My first post"  
    content:"This is some test content"  
    url:"https://aws.amazon.com/appsync/"  
  ){  
    id  
    title  
  }  
}
```

Questo scrive i dati su DynamoDB, che poi trasmette i dati tramite Lambda ad OpenSearch Amazon Service, dove puoi cercare tutti i post in base a campi diversi. Ad esempio, poiché i dati si trovano in Amazon OpenSearch Service, puoi cercare nei campi dell'autore o del contenuto con testo in formato libero, anche con spazi, come segue:

```
query searchName{  
  searchAuthor(name:"  Nadia  "){  
    id  
    title  
    content  
  }  
}  
  
query searchContent{  
  searchContent(text:"test"){  
    id  
    title  
    content  
  }  
}
```

Poiché i dati vengono scritti direttamente su DynamoDB, è comunque possibile eseguire operazioni efficienti di ricerca di elenchi o elementi sulla tabella con le query `and.allPosts{...}` `singlePost{...}` Questo stack utilizza il seguente codice di esempio per i flussi DynamoDB:

Nota: questo codice è solo a scopo esemplificativo.

```
var AWS = require('aws-sdk');
var path = require('path');
var stream = require('stream');

var esDomain = {
  endpoint: 'https://opensearch-domain-name.REGION.es.amazonaws.com',
  region: 'REGION',
  index: 'id',
  doctype: 'post'
};

var endpoint = new AWS.Endpoint(esDomain.endpoint)
var creds = new AWS.EnvironmentCredentials('AWS');

function postDocumentToES(doc, context) {
  var req = new AWS.HttpRequest(endpoint);

  req.method = 'POST';
  req.path = '/_bulk';
  req.region = esDomain.region;
  req.body = doc;
  req.headers['presigned-expires'] = false;
  req.headers['Host'] = endpoint.host;

  // Sign the request (Sigv4)
  var signer = new AWS.Signers.V4(req, 'es');
  signer.addAuthorization(creds, new Date());

  // Post document to ES
  var send = new AWS.NodeHttpClient();
  send.handleRequest(req, null, function (httpResp) {
    var body = '';
    httpResp.on('data', function (chunk) {
      body += chunk;
    });
    httpResp.on('end', function (chunk) {
      console.log('Successful', body);
      context.succeed();
    });
  }, function (err) {
    console.log('Error: ' + err);
    context.fail();
  });
}
```

```

    });
}

exports.handler = (event, context, callback) => {
  console.log("event => " + JSON.stringify(event));
  var posts = '';

  for (var i = 0; i < event.Records.length; i++) {
    var eventName = event.Records[i].eventName;
    var actionType = '';
    var image;
    var noDoc = false;
    switch (eventName) {
      case 'INSERT':
        actionType = 'create';
        image = event.Records[i].dynamodb.NewImage;
        break;
      case 'MODIFY':
        actionType = 'update';
        image = event.Records[i].dynamodb.NewImage;
        break;
      case 'REMOVE':
        actionType = 'delete';
        image = event.Records[i].dynamodb.OldImage;
        noDoc = true;
        break;
    }

    if (typeof image !== "undefined") {
      var postData = {};
      for (var key in image) {
        if (image.hasOwnProperty(key)) {
          if (key === 'postId') {
            postData['id'] = image[key].S;
          } else {
            var val = image[key];
            if (val.hasOwnProperty('S')) {
              postData[key] = val.S;
            } else if (val.hasOwnProperty('N')) {
              postData[key] = val.N;
            }
          }
        }
      }
    }
  }
}

```



```
var action = {};  
action[actionType] = {};  
action[actionType]._index = 'id';  
action[actionType]._type = 'post';  
action[actionType]._id = postData['id'];  
posts += [  
    JSON.stringify(action),  
].concat(noDoc?[]:[JSON.stringify(postData)]).join('\n') + '\n';  
}  
}  
console.log('posts:',posts);  
postDocumentToES(posts, context);  
};
```

Puoi quindi utilizzare i flussi DynamoDB per collegarla a una tabella DynamoDB con una chiave primaria di e qualsiasi modifica all'origine `id` di DynamoDB verrà trasferita nel tuo dominio di servizio. OpenSearch Per ulteriori informazioni sulla configurazione di questa funzionalità, consulta [Documentazione dei flussi di DynamoDB](#).

Utilizzo delle operazioni batch di DynamoDB in AWS AppSync

Note

Ora supportiamo principalmente il runtime `APPSYNC_JS` e la relativa documentazione. [Prendi in considerazione l'utilizzo del runtime APPSYNC_JS e delle relative guide qui.](#)

AWS AppSync supporta l'utilizzo di operazioni batch di Amazon DynamoDB su una o più tabelle in una singola regione. Le operazioni supportate sono `BatchGetItem`, `BatchPutItem` e `BatchDeleteItem`. Usando queste caratteristiche in AWS AppSync, è possibile eseguire attività quali ad esempio:

- Passare un elenco di chiavi in una singola query e restituire i risultati da una tabella
- Leggere i record da una o più tabelle in un'unica query
- Scrivere record in blocco in una o più tabelle
- Scrivere o eliminare in forma condizionata record in più tabelle che possono essere relazionate

L'utilizzo delle operazioni batch con DynamoDB AWS AppSync in è una tecnica avanzata che richiede un po' di riflessione e conoscenza in più delle operazioni di backend e delle strutture delle tabelle. Inoltre, le operazioni in batch AWS AppSync presentano due differenze fondamentali rispetto alle operazioni non in batch:

- Il ruolo dell'origine dati deve disporre delle autorizzazioni per tutte le tabelle a cui il resolver effettua l'accesso.
- La specifica della tabella per un resolver fa parte del modello di mappatura.

Permissions

Come altri resolver, è necessario creare un'origine dati in AWS AppSync e creare un ruolo o utilizzarne uno esistente. Poiché le operazioni batch richiedono autorizzazioni diverse sulle tabelle DynamoDB, è necessario concedere ai ruoli configurati le autorizzazioni per le azioni di lettura o scrittura:

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "dynamodb:BatchGetItem",
        "dynamodb:BatchWriteItem"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:dynamodb:us-east-1:111122223333:table/TABLENAME",
        "arn:aws:dynamodb:us-east-1:111122223333:table/TABLENAME/*"
      ]
    }
  ]
}
```

Nota: i ruoli sono legati alle fonti di dati in AWS AppSync e i resolver sui campi vengono richiamati su un'origine dati. Le fonti di dati configurate per il recupero con DynamoDB hanno solo una tabella specificata, per semplificare la configurazione. Pertanto, quando si esegue un'operazione di batch

su più tabelle in un singolo resolver, che è una delle attività più avanzate, è necessario concedere il ruolo sull'accesso all'origine dati a tutte le tabelle con cui interagirà il resolver. Questo potrebbe essere fatto nel campo **Risorsa** nella policy di IAM indicata in precedenza. Viene effettuata la configurazione delle tabelle per effettuare chiamate in batch nel modello di resolver, che verrà descritto di seguito.

Origine dati

Per semplicità, useremo la stessa origine dati per tutti i resolver usati in questo tutorial. Nella scheda **Sorgenti dati**, crea una nuova origine dati **DynamoDB** e assegna un nome. **BatchTutorial** Per la tabella è possibile specificare un qualsiasi nome, perché i nomi delle tabelle sono specificati come parte del modello di mappatura delle richieste per le operazioni in batch. La tabella si chiamerà **empty**.

Per questo tutorial, funzionerà qualsiasi ruolo con le seguenti policy inline:

Batch a tabella singola

Warning

BatchPutItem e **BatchDeleteItem** non sono supportati se utilizzati per il rilevamento e la risoluzione dei conflitti. Queste impostazioni devono essere disabilitate per evitare possibili errori.

Per questo esempio, si supponga di disporre di una tabella singola denominata **Post** a cui si desidera aggiungere o rimuovere voci con operazioni in batch. Usa lo schema seguente, notando che per la query passeremo un elenco di IDs:

```
type Post {
  id: ID!
  title: String
}

input PostInput {
  id: ID!
  title: String
}

type Query {
```

```

    batchGet(ids: [ID]): [Post]
  }

  type Mutation {
    batchAdd(posts: [PostInput]): [Post]
    batchDelete(ids: [ID]): [Post]
  }

  schema {
    query: Query
    mutation: Mutation
  }

```

Allegare un resolver al campo `batchAdd()` con il seguente Modello di mappatura della richiesta. Questo prende automaticamente ogni voce nel tipo `input PostInput` di GraphQL e crea una mappa, che è necessaria per l'operazione `BatchPutItem`:

```

#set($postsdata = [])
#foreach($item in ${ctx.args.posts})
    $util.qr($postsdata.add($util.dynamodb.toMapValues($item)))
#end

{
  "version" : "2018-05-29",
  "operation" : "BatchPutItem",
  "tables" : {
    "Posts": $utils.toJson($postsdata)
  }
}

```

In questo caso, il Modello di mappatura della risposta è un semplice passaggio, ma notare che il nome della tabella viene aggiunto come `..data.Posts` all'oggetto del contesto come segue:

```
$util.toJson($ctx.result.data.Posts)
```

Ora accedere alla pagina Query della console di AWS AppSync ed eseguire la seguente mutazione `batchAdd`:

```

mutation add {
  batchAdd(posts:[{
    id: 1 title: "Running in the Park"},{
    id: 2 title: "Playing fetch"
  }])
}

```

```

    ])]{
      id
      title
    }
  }
}
```

Dovresti vedere i risultati stampati sullo schermo e puoi convalidare in modo indipendente tramite la console DynamoDB che entrambi i valori sono stati scritti nella tabella Posts.

Successivamente, allegare un resolver al campo `batchGet()` con il seguente Modello di mappatura della richiesta. Questo prende automaticamente ogni voce nel tipo `ids: []` di GraphQL e crea una mappa, che risulta necessaria per l'operazione `BatchGetItem`:

```

#set($ids = [])
#foreach($id in ${ctx.args.ids})
  #set($map = {})
  $util.qr($map.put("id", $util.dynamodb.toString($id)))
  $util.qr($ids.add($map))
#end

{
  "version" : "2018-05-29",
  "operation" : "BatchGetItem",
  "tables" : {
    "Posts": {
      "keys": $util.toJson($ids),
      "consistentRead": true,
      "projection" : {
        "expression" : "#id, title",
        "expressionNames" : { "#id" : "id" }
      }
    }
  }
}
```

Il Modello di mappatura della risposta è ancora un semplice passaggio, con di nuovo il nome della tabella che viene aggiunto come `..data.Posts` all'oggetto del contesto:

```
$util.toJson($ctx.result.data.Posts)
```

Ora tornare alla pagina Query della console di AWS AppSync ed eseguire la seguente Query di `batchAdd`:

```
query get {  
  batchGet(ids:[1,2,3]){  
    id  
    title  
  }  
}
```

Questo dovrebbe restituire i risultati per i due valori `id` aggiunti in precedenza. Si noti che un valore `null` è stato restituito per `id` con un valore di 3. Questo perché non era presente nessun record nella tabella `Post` che avesse ancora tale valore. Tieni inoltre presente che AWS AppSync restituisce i risultati nello stesso ordine delle chiavi passate alla query, una funzionalità aggiuntiva che viene eseguita per conto AWS AppSync tuo. Pertanto, se si passa a `batchGet(ids:[1,3,2])`, si visualizzerà l'ordine modificato. È inoltre possibile sapere quale `id` ha restituito un valore `null`.

Infine, allegare un resolver al campo `batchDelete()` con il seguente Modello di mappatura della richiesta. Questo prende automaticamente ogni voce nel tipo `ids: []` di GraphQL e crea una mappa, che risulta necessaria per l'operazione `BatchGetItem`:

```
#set($ids = [])  
#foreach($id in ${ctx.args.ids})  
  #set($map = {})  
  $util.qr($map.put("id", $util.dynamodb.toString($id)))  
  $util.qr($ids.add($map))  
#end  
  
{  
  "version" : "2018-05-29",  
  "operation" : "BatchDeleteItem",  
  "tables" : {  
    "Posts": $util.toJson($ids)  
  }  
}
```

Il Modello di mappatura della risposta è ancora un semplice passaggio, con di nuovo il nome della tabella che viene aggiunto come `..data.Posts` all'oggetto del contesto:

```
$util.toJson($ctx.result.data.Posts)
```

Ora tornare alla pagina Query della console di AWS AppSync ed eseguire la seguente mutazione `batchDelete`:

```
mutation delete {  
  batchDelete(ids:[1,2]){ id }  
}
```

I record con id 1 e 2 dovrebbero essere eliminati. Se si esegue nuovamente la query `batchGet()` da una versione precedente, questi devono restituire `null`.

Batch a tabella multipla

Warning

`BatchPutItem` e `BatchDeleteItem` non sono supportati se utilizzati per il rilevamento e la risoluzione dei conflitti. Queste impostazioni devono essere disabilitate per evitare possibili errori.

AWS AppSync consente inoltre di eseguire operazioni in batch su più tabelle. L'applicazione seguente è più complessa da costruire. Si immagina di creare un'app per la salute degli animali, in cui i sensori segnalano la posizione dell'animale domestico e la temperatura corporea. I sensori sono dotati di batteria e tentano di connettersi alla rete a distanza di pochi minuti. Quando un sensore stabilisce una connessione, invia le sue letture alla nostra AWS AppSync API. I trigger quindi analizzano i dati in modo da presentare un pannello di controllo al proprietario dell'animale domestico, focalizzando l'attenzione sulla rappresentazione delle interazioni tra il sensore e l'archivio di dati di back-end.

Come prerequisito, creiamo prima due tabelle DynamoDB; `LocationReadings` memorizzerà le letture della posizione del sensore e `TemperatureReadings` memorizzerà le letture della temperatura del sensore. Entrambe le tabelle condividono la stessa struttura della chiave primaria: la chiave di partizione `sensorId` (`String`) e la chiave di ordinamento `timestamp` (`String`).

Utilizzare il seguente schema di GraphQL:

```
type Mutation {  
  # Register a batch of readings  
  recordReadings(tempReadings: [TemperatureReadingInput], locReadings:  
    [LocationReadingInput]): RecordResult  
  # Delete a batch of readings  
  deleteReadings(tempReadings: [TemperatureReadingInput], locReadings:  
    [LocationReadingInput]): RecordResult  
}
```

```
type Query {
  # Retrieve all possible readings recorded by a sensor at a specific time
  getReadings(sensorId: ID!, timestamp: String!): [SensorReading]
}

type RecordResult {
  temperatureReadings: [TemperatureReading]
  locationReadings: [LocationReading]
}

interface SensorReading {
  sensorId: ID!
  timestamp: String!
}

# Sensor reading representing the sensor temperature (in Fahrenheit)
type TemperatureReading implements SensorReading {
  sensorId: ID!
  timestamp: String!
  value: Float
}

# Sensor reading representing the sensor location (lat,long)
type LocationReading implements SensorReading {
  sensorId: ID!
  timestamp: String!
  lat: Float
  long: Float
}

input TemperatureReadingInput {
  sensorId: ID!
  timestamp: String
  value: Float
}

input LocationReadingInput {
  sensorId: ID!
  timestamp: String
  lat: Float
  long: Float
}
```


BatchPutItem - Registrazione delle letture del sensore

I sensori devono essere in grado di inviare le loro letture una volta stabilita la connessione a Internet. Il campo `Mutation.recordReadings` di GraphQL è l'API che si userà per farlo. Allegare un resolver per portare alla vita l'API.

Selezionare **Attach** (Allega) vicino al campo `Mutation.recordReadings`. Sulla schermata successiva, scegliere la stessa origine dati `BatchTutorial` creata all'inizio di questo tutorial.

Aggiungere il seguente modello di mappatura della richiesta.

Modello di mappatura della richiesta

```
## Convert tempReadings arguments to DynamoDB objects
#set($tempReadings = [])
#foreach($reading in ${ctx.args.tempReadings})
    $util.qr($tempReadings.add($util.dynamodb.toMapValues($reading)))
#end

## Convert locReadings arguments to DynamoDB objects
#set($locReadings = [])
#foreach($reading in ${ctx.args.locReadings})
    $util.qr($locReadings.add($util.dynamodb.toMapValues($reading)))
#end

{
  "version" : "2018-05-29",
  "operation" : "BatchPutItem",
  "tables" : {
    "locationReadings": $utils.toJson($locReadings),
    "temperatureReadings": $utils.toJson($tempReadings)
  }
}
```

Come si può vedere, l'operazione `BatchPutItem` consente di specificare più tabelle.

Usare il seguente modello di mappatura della risposta.

Modello di mappatura della risposta

```
## If there was an error with the invocation
## there might have been partial results
```

```

# if($ctx.error)
  ## Append a GraphQL error for that field in the GraphQL response
  $utils.appendError($ctx.error.message, $ctx.error.message)
# end
## Also returns data for the field in the GraphQL response
$utils.toJson($ctx.result.data)

```

Con le operazioni in batch, possono essere presenti entrambi gli errori e i risultati restituiti dalla chiamata. In questo caso, è possibile eseguire liberamente ulteriori operazioni di gestione degli errori.

Nota: l'uso di `$utils.appendError()` è analogo a quello di `$util.error()`, con la differenza principale che non interrompe la valutazione del modello di mappatura. Al contrario, segnala che si è verificato un errore con il campo, ma consente al modello di essere valutato e, di conseguenza, di restituire i dati al chiamante. Si consiglia di usare `$utils.appendError()` quando un'applicazione deve restituire risultati parziali.

Salva il resolver e vai alla pagina Query della console. AWS AppSync Inviare alcune letture del sensore.

Eseguire la mutazione seguente:

```

mutation sendReadings {
  recordReadings(
    tempReadings: [
      {sensorId: 1, value: 85.5, timestamp: "2018-02-01T17:21:05.000+08:00"},
      {sensorId: 1, value: 85.7, timestamp: "2018-02-01T17:21:06.000+08:00"},
      {sensorId: 1, value: 85.8, timestamp: "2018-02-01T17:21:07.000+08:00"},
      {sensorId: 1, value: 84.2, timestamp: "2018-02-01T17:21:08.000+08:00"},
      {sensorId: 1, value: 81.5, timestamp: "2018-02-01T17:21:09.000+08:00"}
    ]
    locReadings: [
      {sensorId: 1, lat: 47.615063, long: -122.333551, timestamp:
"2018-02-01T17:21:05.000+08:00"},
      {sensorId: 1, lat: 47.615163, long: -122.333552, timestamp:
"2018-02-01T17:21:06.000+08:00"},
      {sensorId: 1, lat: 47.615263, long: -122.333553, timestamp:
"2018-02-01T17:21:07.000+08:00"},
      {sensorId: 1, lat: 47.615363, long: -122.333554, timestamp:
"2018-02-01T17:21:08.000+08:00"},
      {sensorId: 1, lat: 47.615463, long: -122.333555, timestamp:
"2018-02-01T17:21:09.000+08:00"}
    ]) {

```

```

    locationReadings {
      sensorId
      timestamp
      lat
      long
    }
    temperatureReadings {
      sensorId
      timestamp
      value
    }
  }
}

```

Sono state inviate 10 letture del sensore in una sola mutazione, con letture suddivise in due tabelle. Utilizza la console DynamoDB per verificare che i dati vengano visualizzati nelle tabelle LocationReadings e Temperaturereadings.

BatchDeleteItem - Eliminazione delle letture del sensore

Analogamente, sarebbe necessario anche eliminare i batch delle letture del sensore. Utilizzare il campo `Mutation.deleteReadings` di GraphQL per questo scopo. Selezionare Attach (Allega) vicino al campo `Mutation.recordReadings`. Sulla schermata successiva, scegliere la stessa origine dati BatchTutorial1 creata all'inizio di questo tutorial.

Utilizzare il seguente modello di mappatura della richiesta.

Modello di mappatura della richiesta

```

## Convert tempReadings arguments to DynamoDB primary keys
#set($tempReadings = [])
#foreach($reading in ${ctx.args.tempReadings})
  #set($pkey = {})
  $util.qr($pkey.put("sensorId", $reading.sensorId))
  $util.qr($pkey.put("timestamp", $reading.timestamp))
  $util.qr($tempReadings.add($util.dynamodb.toMapValues($pkey)))
#end

## Convert locReadings arguments to DynamoDB primary keys
#set($locReadings = [])
#foreach($reading in ${ctx.args.locReadings})
  #set($pkey = {})

```

```

    $util.qr($pkey.put("sensorId", $reading.sensorId))
    $util.qr($pkey.put("timestamp", $reading.timestamp))
    $util.qr($locReadings.add($util.dynamodb.toMapValues($pkey)))
#end

{
  "version" : "2018-05-29",
  "operation" : "BatchDeleteItem",
  "tables" : {
    "locationReadings": $utils.toJson($locReadings),
    "temperatureReadings": $utils.toJson($tempReadings)
  }
}

```

Il modello di mappatura della risposta è lo stesso utilizzato per `Mutation.recordReadings`.

Modello di mappatura della risposta

```

## If there was an error with the invocation
## there might have been partial results
#if($ctx.error)
  ## Append a GraphQL error for that field in the GraphQL response
  $utils.appendError($ctx.error.message, $ctx.error.message)
#end
## Also return data for the field in the GraphQL response
$utils.toJson($ctx.result.data)

```

Salva il resolver e vai alla pagina Query della console. AWS AppSync Ora, eliminare un paio di letture del sensore.

Eseguire la mutazione seguente:

```

mutation deleteReadings {
  # Let's delete the first two readings we recorded
  deleteReadings(
    tempReadings: [{sensorId: 1, timestamp: "2018-02-01T17:21:05.000+08:00"}]
    locReadings: [{sensorId: 1, timestamp: "2018-02-01T17:21:05.000+08:00"}]) {
    locationReadings {
      sensorId
      timestamp
      lat
      long
    }
  }
}

```

```

    }
    temperatureReadings {
      sensorId
      timestamp
      value
    }
  }
}

```

Verifica tramite la console DynamoDB che queste due letture siano state eliminate dalle tabelle `LocationReadings` e `TemperatureReadings`.

BatchGetItem - Recupera le letture

Un'altra operazione comune per l'app Pet Health sarebbe quella di recuperare le letture per un sensore in un determinato momento. Allegare un resolver al campo `Query.getReadings` di GraphQL sullo schema. Selezionare Attach (Allega) e, sulla schermata successiva, scegliere la stessa origine dati `BatchTutorial` creata all'inizio del tutorial.

Aggiungere il seguente modello di mappatura della richiesta.

Modello di mappatura della richiesta

```

## Build a single DynamoDB primary key,
## as both locationReadings and tempReadings tables
## share the same primary key structure
#set($pkey = {})
$util.qr($pkey.put("sensorId", $ctx.args.sensorId))
$util.qr($pkey.put("timestamp", $ctx.args.timestamp))

{
  "version" : "2018-05-29",
  "operation" : "BatchGetItem",
  "tables" : {
    "locationReadings": {
      "keys": [$util.dynamodb.toMapValuesJson($pkey)],
      "consistentRead": true
    },
    "temperatureReadings": {
      "keys": [$util.dynamodb.toMapValuesJson($pkey)],
      "consistentRead": true
    }
  }
}

```

```

    }
  }
}

```

Nota che ora stiamo usando l'BatchGetItemoperazione.

Il modello di mappatura della risposta sarà leggermente diverso perché è stato scelto di restituire un elenco `SensorReading`. Mappare il risultato della chiamata nella forma desiderata.

Modello di mappatura della risposta

```

## Merge locationReadings and temperatureReadings
## into a single list
## __typename needed as schema uses an interface
#set($sensorReadings = [])

#foreach($locReading in $ctx.result.data.locationReadings)
    $util.qr($locReading.put("__typename", "LocationReading"))
    $util.qr($sensorReadings.add($locReading))
#end

#foreach($tempReading in $ctx.result.data.temperatureReadings)
    $util.qr($tempReading.put("__typename", "TemperatureReading"))
    $util.qr($sensorReadings.add($tempReading))
#end

$util.toJson($sensorReadings)

```

Salva il resolver e vai alla pagina Query della console. AWS AppSync Ora, recuperare le letture del sensore.

Eseguire la query seguente:

```

query getReadingsForSensorAndTime {
  # Let's retrieve the very first two readings
  getReadings(sensorId: 1, timestamp: "2018-02-01T17:21:06.000+08:00") {
    sensorId
    timestamp
    ...on TemperatureReading {
      value
    }
    ...on LocationReading {
      lat
    }
  }
}

```

```
    long
  }
}
```

Abbiamo dimostrato con successo l'uso delle operazioni batch di DynamoDB utilizzando AWS AppSync

Gestione errori

Nel AWS AppSync, le operazioni sulle sorgenti dati a volte possono restituire risultati parziali. Risultati parziali è il termine che si userà per indicare quando l'output di un'operazione comprende alcuni dati e un errore. Poiché la gestione degli errori è intrinsecamente specifica dell'applicazione, AWS AppSync offre la possibilità di gestire gli errori nel modello di mappatura della risposta. L'errore di chiamata del resolver, se presente, è disponibile nel contesto come `$ctx.error`. Gli errori di chiamata comprendono sempre un messaggio e un tipo, accessibili come proprietà `$ctx.error.message` e `$ctx.error.type`. Durante la chiamata del modello di mappatura della risposta, è possibile gestire i risultati parziali in tre modi:

1. assumere l'errore di chiamata semplicemente restituendo i dati
2. generare un errore (usando `$util.error(...)`) arrestando la valutazione del modello di mappatura della risposta, che non restituisce nessun dato.
3. accodare un errore (usando `$util.appendError(...)`) e restituire anche i dati

Ora è opportuno dimostrare ciascuno dei tre punti sopra indicati con le operazioni in batch di DynamoDB.

Operazioni di batch di DynamoDB

Con le operazioni di batch di DynamoDB, è possibile che un batch sia completato parzialmente. Ovvero, è possibile che alcune delle chiavi o voci richieste non vengano elaborate. Se non AWS AppSync è in grado di completare un batch, nel contesto verranno impostati elementi non elaborati e un errore di invocazione.

La gestione degli errori verrà realizzata tramite la configurazione del campo `Query.getReadings` dall'operazione `BatchGetItem` dalla sezione precedente di questo tutorial. In questo momento, è opportuno supporre che durante l'esecuzione del campo `Query.getReadings`, la tabella `temperatureReadings` DynamoDB esegua un throughput assegnato. DynamoDB ha generato

ProvisionedThroughputExceededException un al secondo tentativo AWS AppSync per elaborare gli elementi rimanenti del batch.

Il seguente JSON rappresenta il contesto serializzato dopo la chiamata del batch di DynamoDB prima che il modello di mappatura della risposta venisse valutato.

```
{
  "arguments": {
    "sensorId": "1",
    "timestamp": "2018-02-01T17:21:05.000+08:00"
  },
  "source": null,
  "result": {
    "data": {
      "temperatureReadings": [
        null
      ],
      "locationReadings": [
        {
          "lat": 47.615063,
          "long": -122.333551,
          "sensorId": "1",
          "timestamp": "2018-02-01T17:21:05.000+08:00"
        }
      ]
    },
    "unprocessedKeys": {
      "temperatureReadings": [
        {
          "sensorId": "1",
          "timestamp": "2018-02-01T17:21:05.000+08:00"
        }
      ],
      "locationReadings": []
    }
  },
  "error": {
    "type": "DynamoDB:ProvisionedThroughputExceededException",
    "message": "You exceeded your maximum allowed provisioned throughput for a table or for one or more global secondary indexes. (...)"
  },
  "outErrors": []
}
```


Alcune cose da tenere presente in questo contesto:

- l'errore di invocazione è stato impostato nel contesto in **\$ctx.error** by AWS AppSync e il tipo di errore è stato impostato su DynamoDB: `ProvisionedThroughputExceededException`
- i risultati sono mappati per tabella in `$ctx.result.data`, anche se è presente un errore
- le chiavi che non sono state elaborate sono disponibili in `$ctx.result.data.unprocessedKeys`. Qui, non AWS AppSync è stato possibile recuperare l'elemento con la chiave (SensorID:1, timestamp:2018-02-01T 17:21:05.000 + 08:00) a causa dell'insufficiente velocità di trasmissione della tabella.

Nota: per `BatchPutItem`, è `$ctx.result.data.unprocessedItems`. Per `BatchDeleteItem`, è `$ctx.result.data.unprocessedKeys`.

È possibile gestire l'errore in tre modi diversi.

1. Assumere l'errore di chiamata

Restituire i dati senza gestire l'errore di chiamata assume efficacemente l'errore. In questo modo, il risultato per il campo di GraphQL è sempre positivo.

Il modello di mappatura della risposta che viene scritto è familiare e si focalizza solo sui dati del risultato.

Modello di mappatura della risposta:

```
$util.toJson($ctx.result.data)
```

Risposta di GraphQL:

```
{
  "data": {
    "getReadings": [
      {
        "sensorId": "1",
        "timestamp": "2018-02-01T17:21:05.000+08:00",
        "lat": 47.615063,
        "long": -122.333551
      },
      {
        "sensorId": "1",
```

```

        "timestamp": "2018-02-01T17:21:05.000+08:00",
        "value": 85.5
      }
    ]
  }
}

```

Non verrà aggiunto nessun errore alla risposta di errore poiché sono stati eseguiti solo i dati.

2. Generazione di un errore per interrompere l'esecuzione del modello

Quando i guasti parziali devono essere trattati come guasti totali dalla prospettiva del client, è possibile interrompere l'esecuzione del modello per evitare di restituire i dati. Il metodo di utilità `$util.error(...)` raggiunge esattamente questo comportamento.

Modello di mappatura della risposta:

```

## there was an error let's mark the entire field
## as failed and do not return any data back in the response
#if ($ctx.error)
    $util.error($ctx.error.message, $ctx.error.type, null,
    $ctx.result.data.unprocessedKeys)
#end

$util.toJson($ctx.result.data)

```

Risposta di GraphQL:

```

{
  "data": {
    "getReadings": null
  },
  "errors": [
    {
      "path": [
        "getReadings"
      ],
      "data": null,
      "errorType": "DynamoDB:ProvisionedThroughputExceededException",
      "errorInfo": {
        "temperatureReadings": [
          {

```

```

        "sensorId": "1",
        "timestamp": "2018-02-01T17:21:05.000+08:00"
      }
    ],
    "locationReadings": []
  },
  "locations": [
    {
      "line": 58,
      "column": 3
    }
  ],
  "message": "You exceeded your maximum allowed provisioned throughput for a table
or for one or more global secondary indexes. (...)"
}
]
}

```

Sebbene alcuni risultati possano essere stati restituiti dall'operazione in batch di DynamoDB, si è scelto di generare un errore in modo tale che il campo `getReadings` di GraphQL sia nullo e l'errore venga aggiunto al blocco di errori della risposta GraphQL.

3. Aggiunta di un errore per restituire sia i dati sia gli errori

In alcuni casi, per fornire una migliore esperienza utente, le applicazioni possono restituire risultati parziali e notificare ai client le voci non elaborate. I client possono decidere di implementare un nuovo tentativo oppure di rimandare l'errore all'utente finale. Il `$util.appendError(...)` è il metodo di utilità che consente questo comportamento, permettendo al designer dell'applicazione di accodare gli errori nel contesto senza interferire con la valutazione del modello. Dopo aver valutato il modello, AWS AppSync elaborerà eventuali errori di contesto aggiungendoli al blocco `errors` della risposta GraphQL.

Modello di mappatura della risposta:

```

#if ($ctx.error)
  ## pass the unprocessed keys back to the caller via the `errorInfo` field
  $util.appendError($ctx.error.message, $ctx.error.type, null,
    $ctx.result.data.unprocessedKeys)
#end

$util.toJson($ctx.result.data)

```

Sono stati inoltrati sia l'errore di chiamata sia l'elemento `unprocessedKeys` all'interno del blocco di errori della risposta di GraphQL. Anche il campo `getReadings` restituisce dati parziali dalla tabella `locationReadings` come è possibile vedere nella risposta di seguito.

Risposta di GraphQL:

```
{
  "data": {
    "getReadings": [
      null,
      {
        "sensorId": "1",
        "timestamp": "2018-02-01T17:21:05.000+08:00",
        "value": 85.5
      }
    ]
  },
  "errors": [
    {
      "path": [
        "getReadings"
      ],
      "data": null,
      "errorType": "DynamoDB:ProvisionedThroughputExceededException",
      "errorInfo": {
        "temperatureReadings": [
          {
            "sensorId": "1",
            "timestamp": "2018-02-01T17:21:05.000+08:00"
          }
        ],
        "locationReadings": []
      },
      "locations": [
        {
          "line": 58,
          "column": 3
        }
      ],
      "message": "You exceeded your maximum allowed provisioned throughput for a table or for one or more global secondary indexes. (...)"
    }
  ]
}
```

```
}
```

Esecuzione di transazioni DynamoDB in AWS AppSync

Note

Ora supportiamo principalmente il runtime APPSYNC_JS e la relativa documentazione. [Prendi in considerazione l'utilizzo del runtime APPSYNC_JS e delle relative guide qui.](#)

AWS AppSync supporta l'utilizzo di operazioni di transazione Amazon DynamoDB su una o più tabelle in una singola regione. Le operazioni supportate sono `TransactGetItems` e `TransactWriteItems`. Utilizzando queste funzionalità in AWS AppSync, puoi eseguire attività come:

- Passare un elenco di chiavi in una singola query e restituire i risultati da una tabella
- Leggere i record da una o più tabelle in un'unica query
- Scrivi i record della transazione su una o più tabelle in qualsiasi all-or-nothing modo
- Eseguire transazioni quando alcune condizioni sono soddisfatte

Permissions

Come altri resolver, è necessario creare un'origine dati in AWS AppSync e creare un ruolo o utilizzarne uno esistente. Poiché le operazioni di transazione richiedono autorizzazioni diverse sulle tabelle DynamoDB, è necessario concedere ai ruoli configurati le autorizzazioni per le azioni di lettura o scrittura:

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "dynamodb:DeleteItem",
        "dynamodb:GetItem",
```

```

        "dynamodb:PutItem",
        "dynamodb:Query",
        "dynamodb:Scan",
        "dynamodb:UpdateItem"
    ],
    "Effect": "Allow",
    "Resource": [
        "arn:aws:dynamodb:us-east-1:111122223333:table/TABLENAME",
        "arn:aws:dynamodb:us-east-1:111122223333:table/TABLENAME/*"
    ]
}
]
}
```

Nota: i ruoli sono legati alle fonti di dati in AWS AppSync e i resolver sui campi vengono richiamati su un'origine dati. Le fonti di dati configurate per il recupero con DynamoDB hanno solo una tabella specificata, per semplificare la configurazione. Pertanto, quando si esegue un'operazione di transazione su più tabelle in un singolo resolver, che è una delle attività più avanzate, è necessario concedere il ruolo sull'accesso all'origine dati a tutte le tabelle con cui interagirà il resolver. Questo potrebbe essere fatto nel campo **Risorsa** nella policy di IAM indicata in precedenza. Viene effettuata la configurazione delle chiamate di transazione nelle tabelle nel modello di resolver, che verrà descritto di seguito.

Origine dati

Per semplicità, useremo la stessa origine dati per tutti i resolver usati in questo tutorial. Nella scheda **Sorgenti dati**, crea una nuova origine dati DynamoDB e assegnale un nome. TransactTutorial Per la tabella è possibile specificare un qualsiasi nome, perché i nomi delle tabelle sono specificati come parte del modello di mappatura delle richieste per le operazioni di transazione. La tabella si chiamerà `empty`.

Avremo due tabelle denominate `SavingAccounts` e `CheckingAccounts`, entrambe con `accountNumber` come chiave di partizione e una tabella `transactionHistory` con `transactionId` come chiave di partizione.

Per questo tutorial, funzionerà qualsiasi ruolo con le seguenti policy inline: Sostituisci `region` e `accountId` con la tua regione e l'ID dell'account:

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "dynamodb:DeleteItem",
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:Query",
        "dynamodb:Scan",
        "dynamodb:UpdateItem"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:dynamodb:us-east-1:111122223333:table/savingAccounts",
        "arn:aws:dynamodb:us-east-1:111122223333:table/savingAccounts/*",
        "arn:aws:dynamodb:us-east-1:111122223333:table/checkingAccounts",
        "arn:aws:dynamodb:us-east-1:111122223333:table/checkingAccounts/*",
        "arn:aws:dynamodb:us-east-1:111122223333:table/transactionHistory",
        "arn:aws:dynamodb:us-east-1:111122223333:table/transactionHistory/*"
      ]
    }
  ]
}
```

Transazioni

Per questo esempio, il contesto è una classica transazione bancaria, in cui useremo `TransactWriteItems` per:

- Trasferire denaro dai conti di deposito ai conti correnti
- Generare nuovi record di transazione per ogni transazione

Quindi, useremo `TransactGetItems` per recuperare i dettagli dai conti di deposito ai conti correnti.

 Warning

`TransactWriteItems` non è supportato se utilizzato per il rilevamento e la risoluzione dei conflitti. Queste impostazioni devono essere disabilitate per evitare possibili errori.

Definiamo il nostro schema GraphQL come segue:

```
type SavingAccount {
  accountNumber: String!
  username: String
  balance: Float
}

type CheckingAccount {
  accountNumber: String!
  username: String
  balance: Float
}

type TransactionHistory {
  transactionId: ID!
  from: String
  to: String
  amount: Float
}

type TransactionResult {
  savingAccounts: [SavingAccount]
  checkingAccounts: [CheckingAccount]
  transactionHistory: [TransactionHistory]
}

input SavingAccountInput {
  accountNumber: String!
  username: String
  balance: Float
}

input CheckingAccountInput {
  accountNumber: String!
  username: String
  balance: Float
}
```



```

}

input TransactionInput {
  savingAccountNumber: String!
  checkingAccountNumber: String!
  amount: Float!
}

type Query {
  getAccounts(savingAccountNumbers: [String], checkingAccountNumbers: [String]):
  TransactionResult
}

type Mutation {
  populateAccounts(savingAccounts: [SavingAccountInput], checkingAccounts:
  [CheckingAccountInput]): TransactionResult
  transferMoney(transactions: [TransactionInput]): TransactionResult
}

schema {
  query: Query
  mutation: Mutation
}

```

TransactWriteItems - Compila gli account

Al fine di trasferire denaro tra gli account, abbiamo bisogno di popolare la tabella con i dettagli. Per farlo, useremo l'operazione GraphQL `Mutation.populateAccounts`.

Nella sezione Schema, fai clic su [Allega accanto all'Mutation.populateAccounts](#) operazione. Vai a [VTL Unit Resolvers](#), quindi scegli la stessa fonte di dati. `TransactTutorial`

Usa modello di mappatura della richiesta seguente:

Modello di mappatura della richiesta

```

#set($savingAccountTransactPutItems = [])
#set($index = 0)
#foreach($savingAccount in ${ctx.args.savingAccounts})
  #set($keyMap = {})
  $util.qr($keyMap.put("accountNumber",
  $util.dynamodb.toString($savingAccount.accountNumber)))
  #set($attributeValues = {})

```

```

    $util.qr($attributeValues.put("username",
$util.dynamodb.toString($savingAccount.username)))
    $util.qr($attributeValues.put("balance",
$util.dynamodb.toNumber($savingAccount.balance)))
    #set($index = $index + 1)
    #set($savingAccountTransactPutItem = {"table": "savingAccounts",
      "operation": "PutItem",
      "key": $keyMap,
      "attributeValues": $attributeValues})
    $util.qr($savingAccountTransactPutItems.add($savingAccountTransactPutItem))
#end

#set($checkingAccountTransactPutItems = [])
#set($index = 0)
#foreach($checkingAccount in ${ctx.args.checkingAccounts})
    #set($keyMap = {})
    $util.qr($keyMap.put("accountNumber",
$util.dynamodb.toString($checkingAccount.accountNumber)))
    #set($attributeValues = {})
    $util.qr($attributeValues.put("username",
$util.dynamodb.toString($checkingAccount.username)))
    $util.qr($attributeValues.put("balance",
$util.dynamodb.toNumber($checkingAccount.balance)))
    #set($index = $index + 1)
    #set($checkingAccountTransactPutItem = {"table": "checkingAccounts",
      "operation": "PutItem",
      "key": $keyMap,
      "attributeValues": $attributeValues})
    $util.qr($checkingAccountTransactPutItems.add($checkingAccountTransactPutItem))
#end

#set($transactItems = [])
$util.qr($transactItems.addAll($savingAccountTransactPutItems))
$util.qr($transactItems.addAll($checkingAccountTransactPutItems))

{
  "version" : "2018-05-29",
  "operation" : "TransactWriteItems",
  "transactItems" : $util.toJson($transactItems)
}

```

E il seguente modello di mappatura della risposta:

Modello di mappatura della risposta

```

#if ($ctx.error)
    $util.appendError($ctx.error.message, $ctx.error.type, null,
    $ctx.result.cancellationReasons)
#end

#set($savingAccounts = [])
#foreach($index in [0..2])
    $util.qr($savingAccounts.add(${ctx.result.keys[$index]}))
#end

#set($checkingAccounts = [])
#foreach($index in [3..5])
    $util.qr($checkingAccounts.add(${ctx.result.keys[$index]}))
#end

#set($transactionResult = {})
$util.qr($transactionResult.put('savingAccounts', $savingAccounts))
$util.qr($transactionResult.put('checkingAccounts', $checkingAccounts))

$util.toJson($transactionResult)

```

Salva il resolver e vai alla sezione Queries della console per popolare gli account. AWS AppSync

Eseguire la mutazione seguente:

```

mutation populateAccounts {
  populateAccounts (
    savingAccounts: [
      {accountNumber: "1", username: "Tom", balance: 100},
      {accountNumber: "2", username: "Amy", balance: 90},
      {accountNumber: "3", username: "Lily", balance: 80},
    ]
    checkingAccounts: [
      {accountNumber: "1", username: "Tom", balance: 70},
      {accountNumber: "2", username: "Amy", balance: 60},
      {accountNumber: "3", username: "Lily", balance: 50},
    ]) {
    savingAccounts {
      accountNumber
    }
    checkingAccounts {
      accountNumber
    }
  }
}

```

```

    }
  }

```

Abbiamo popolato 3 conti di deposito e 3 conti correnti in una mutazione.

Utilizza la console DynamoDB per verificare che i dati vengano visualizzati nelle tabelle `SavingAccounts` e `CheckingAccounts`.

TransactWriteItems - Trasferimento di denaro

Allegare un resolver alla mutazione `transferMoney` con il seguente Modello di mappatura della richiesta. Tenere presente che i valori di `amounts`, `savingAccountNumbers` e `checkingAccountNumbers` sono gli stessi.

```

#set($amounts = [])
#foreach($transaction in ${ctx.args.transactions})
    #set($attributeValueMap = {})
    $util.qr($attributeValueMap.put(":amount",
    $util.dynamodb.toNumber($transaction.amount)))
    $util.qr($amounts.add($attributeValueMap))
#end

#set($savingAccountTransactUpdateItems = [])
#set($index = 0)
#foreach($transaction in ${ctx.args.transactions})
    #set($keyMap = {})
    $util.qr($keyMap.put("accountNumber",
    $util.dynamodb.toString($transaction.savingAccountNumber)))
    #set($update = {})
    $util.qr($update.put("expression", "SET balance = balance - :amount"))
    $util.qr($update.put("expressionValues", $amounts[$index]))
    #set($index = $index + 1)
    #set($savingAccountTransactUpdateItem = {"table": "savingAccounts",
    "operation": "UpdateItem",
    "key": $keyMap,
    "update": $update})
    $util.qr($savingAccountTransactUpdateItems.add($savingAccountTransactUpdateItem))
#end

#set($checkingAccountTransactUpdateItems = [])
#set($index = 0)
#foreach($transaction in ${ctx.args.transactions})
    #set($keyMap = {})

```

```

    $util.qr($keyMap.put("accountNumber",
$util.dynamodb.toString($transaction.checkingAccountNumber)))
    #set($update = {})
    $util.qr($update.put("expression", "SET balance = balance + :amount"))
    $util.qr($update.put("expressionValues", $amounts[$index]))
    #set($index = $index + 1)
    #set($checkingAccountTransactUpdateItem = {"table": "checkingAccounts",
        "operation": "UpdateItem",
        "key": $keyMap,
        "update": $update})

    $util.qr($checkingAccountTransactUpdateItems.add($checkingAccountTransactUpdateItem))
#end

#set($transactionHistoryTransactPutItems = [])
#foreach($transaction in ${ctx.args.transactions})
    #set($keyMap = {})
    $util.qr($keyMap.put("transactionId", $util.dynamodb.toString(${utils.autoId()})))
    #set($attributeValues = {})
    $util.qr($attributeValues.put("from",
$util.dynamodb.toString($transaction.savingAccountNumber)))
    $util.qr($attributeValues.put("to",
$util.dynamodb.toString($transaction.checkingAccountNumber)))
    $util.qr($attributeValues.put("amount",
$util.dynamodb.toNumber($transaction.amount)))
    #set($transactionHistoryTransactPutItem = {"table": "transactionHistory",
        "operation": "PutItem",
        "key": $keyMap,
        "attributeValues": $attributeValues})

    $util.qr($transactionHistoryTransactPutItems.add($transactionHistoryTransactPutItem))
#end

#set($transactItems = [])
$util.qr($transactItems.addAll($savingAccountTransactUpdateItems))
$util.qr($transactItems.addAll($checkingAccountTransactUpdateItems))
$util.qr($transactItems.addAll($transactionHistoryTransactPutItems))

{
    "version" : "2018-05-29",
    "operation" : "TransactWriteItems",
    "transactItems" : $util.toJson($transactItems)
}

```

Avremo 3 transazioni bancarie in una singola operazione `TransactWriteItems`. Utilizzare il seguente modello di mappatura della risposta:

```
#if ($ctx.error)
    $util.appendError($ctx.error.message, $ctx.error.type, null,
    $ctx.result.cancellationReasons)
#end

#set($savingAccounts = [])
#foreach($index in [0..2])
    $util.qr($savingAccounts.add(${ctx.result.keys[$index]}))
#end

#set($checkingAccounts = [])
#foreach($index in [3..5])
    $util.qr($checkingAccounts.add(${ctx.result.keys[$index]}))
#end

#set($transactionHistory = [])
#foreach($index in [6..8])
    $util.qr($transactionHistory.add(${ctx.result.keys[$index]}))
#end

#set($transactionResult = {})
$util.qr($transactionResult.put('savingAccounts', $savingAccounts))
$util.qr($transactionResult.put('checkingAccounts', $checkingAccounts))
$util.qr($transactionResult.put('transactionHistory', $transactionHistory))

$util.toJson($transactionResult)
```

Passare ora alla sezione Query della console AWS AppSync ed eseguire la mutazione `transferMoney` come segue:

```
mutation write {
  transferMoney(
    transactions: [
      {savingAccountNumber: "1", checkingAccountNumber: "1", amount: 7.5},
      {savingAccountNumber: "2", checkingAccountNumber: "2", amount: 6.0},
      {savingAccountNumber: "3", checkingAccountNumber: "3", amount: 3.3}
    ] {
    savingAccounts {
      accountNumber
    }
  }
}
```

```

    checkingAccounts {
      accountNumber
    }
    transactionHistory {
      transactionId
    }
  }
}

```

Abbiamo inviato 2 transazioni bancarie in una mutazione. Utilizza la console DynamoDB per verificare che i dati vengano visualizzati nelle tabelle SavingAccounts, CheckingAccounts e TransactionHistory.

TransactGetItems - Recupera account

Al fine di recuperare i dettagli dai conti di deposito ai conti correnti in una singola richiesta transazionale, allegheremo un resolver all'operazione GraphQL Query .getAccounts sul nostro schema. Seleziona Allega, vai a VTL Unit Resolvers, quindi nella schermata successiva, scegli la stessa fonte di TransactTutorial1 dati creata all'inizio del tutorial. Configurare i modelli come segue:

Modello di mappatura della richiesta

```

#set($savingAccountsTransactGets = [])
#foreach($savingAccountNumber in ${ctx.args.savingAccountNumbers})
    #set($savingAccountKey = {})
    $util.qr($savingAccountKey.put("accountNumber",
    $util.dynamodb.toString($savingAccountNumber)))
    #set($savingAccountTransactGet = {"table": "savingAccounts", "key":
    $savingAccountKey})
    $util.qr($savingAccountsTransactGets.add($savingAccountTransactGet))
#end

#set($checkingAccountsTransactGets = [])
#foreach($checkingAccountNumber in ${ctx.args.checkingAccountNumbers})
    #set($checkingAccountKey = {})
    $util.qr($checkingAccountKey.put("accountNumber",
    $util.dynamodb.toString($checkingAccountNumber)))
    #set($checkingAccountTransactGet = {"table": "checkingAccounts", "key":
    $checkingAccountKey})
    $util.qr($checkingAccountsTransactGets.add($checkingAccountTransactGet))
#end

```

```
#set($transactItems = [])
$util.qr($transactItems.addAll($savingAccountsTransactGets))
$util.qr($transactItems.addAll($checkingAccountsTransactGets))

{
  "version" : "2018-05-29",
  "operation" : "TransactGetItems",
  "transactItems" : $util.toJson($transactItems)
}
```

Modello di mappatura della risposta

```
#if ($ctx.error)
  $util.appendError($ctx.error.message, $ctx.error.type, null,
    $ctx.result.cancellationReasons)
#end

#set($savingAccounts = [])
#foreach($index in [0..2])
  $util.qr($savingAccounts.add($ctx.result.items[$index]))
#end

#set($checkingAccounts = [])
#foreach($index in [3..4])
  $util.qr($checkingAccounts.add($ctx.result.items[$index]))
#end

#set($transactionResult = {})
$util.qr($transactionResult.put('savingAccounts', $savingAccounts))
$util.qr($transactionResult.put('checkingAccounts', $checkingAccounts))

$util.toJson($transactionResult)
```

Salva il resolver e vai alle sezioni Queries della console. AWS AppSync Per recuperare i conti di deposito ai conti correnti, eseguire la seguente query:

```
query getAccounts {
  getAccounts(
    savingAccountNumbers: ["1", "2", "3"],
    checkingAccountNumbers: ["1", "2"]
  ) {
    savingAccounts {
```



```
    accountNumber
    username
    balance
  }
  checkingAccounts {
    accountNumber
    username
    balance
  }
}
```

Abbiamo dimostrato con successo l'uso delle transazioni DynamoDB utilizzando AWS AppSync

Utilizzo di resolver HTTP in AWS AppSync

Note

Ora supportiamo principalmente il runtime APPSYNC_JS e la relativa documentazione.

[Prendi in considerazione l'utilizzo del runtime APPSYNC_JS e delle relative guide qui.](#)

AWS AppSync consente di utilizzare fonti di dati supportate (ad esempio Amazon DynamoDB AWS Lambda, Amazon Service o OpenSearch Amazon Aurora) per eseguire varie operazioni, oltre a qualsiasi endpoint HTTP arbitrario per risolvere i campi GraphQL. Quando gli endpoint HTTP sono disponibili, è possibile connettersi a questi utilizzando un'origine dati. Quindi, è possibile configurare un resolver nello schema per eseguire operazioni GraphQL, ad esempio query, mutazioni e sottoscrizioni. Questo tutorial fornirà una guida di alcuni esempi comuni.

In questo tutorial utilizzi un'API REST (creata utilizzando Amazon API Gateway e Lambda) con un endpoint GraphQL AWS AppSync .

Impostazione One-Click

Se desideri configurare automaticamente un endpoint GraphQL AWS AppSync con un endpoint HTTP configurato (utilizzando Amazon API Gateway e Lambda), puoi utilizzare il seguente modello: AWS CloudFormation

Launch Stack



Creazione di un'API REST

Puoi utilizzare il seguente AWS CloudFormation modello per configurare un endpoint REST adatto a questo tutorial:



Lo AWS CloudFormation stack esegue i seguenti passaggi:

1. Imposta una funzione Lambda che contiene la logica di business per il tuo microservizio.
2. Imposta un'API REST API Gateway con la seguente combinazione endpoint/method/content di tipi:

Percorso risorsa API	Metodo HTTP	Tipo di contenuto supportato
/v1/utenti	POST	application/json
/v1/utenti	GET	application/json
/v1/utenti/1	GET	application/json
/v1/utenti/1	PUT	application/json
/v1/utenti/1	DELETE	application/json

Creazione dell'API GraphQL

Per creare l'API GraphQL in: AWS AppSync

- Apri la AWS AppSync console e scegli Crea API.
- Per il nome API, digita UserData.
- Scegli Custom schema (Schema personalizzato).
- Scegli Create (Crea).

La AWS AppSync console crea una nuova API GraphQL per te utilizzando la modalità di autenticazione della chiave API. Puoi usare la console per configurare ulteriormente l'API GraphQL ed eseguire query sull'API per il resto di questo tutorial.

Creazione di uno schema GraphQL

Ora che hai un'API GraphQL, è possibile creare uno schema GraphQL. Dall'editor dello schema nella AWS AppSync console, assicurati che lo schema corrisponda al seguente schema:

```
schema {  
  query: Query  
  mutation: Mutation  
}  
  
type Mutation {  
  addUser(userInput: UserInput!): User  
  deleteUser(id: ID!): User  
}  
  
type Query {  
  getUser(id: ID): User  
  listUser: [User!]!  
}  
  
type User {  
  id: ID!  
  username: String!  
  firstname: String  
  lastname: String  
  phone: String  
  email: String  
}  
  
input UserInput {  
  id: ID!  
  username: String!  
  firstname: String  
  lastname: String  
  phone: String  
  email: String  
}
```

Configurazione dell'origine dati HTTP

Per configurare l'origine dati HTTP, effettua le seguenti operazioni:

- Nella DataSource scheda, scegli Nuovo, quindi digita un nome descrittivo per l'origine dati (ad esempio, HTTP).
- In Data source type (Tipo di origine dati), scegliere HTTP.
- Imposta l'endpoint sull'endpoint API Gateway che viene creato. Assicurarsi di non includere il nome della fase come parte dell'endpoint.

Nota: al momento solo gli endpoint pubblici sono supportati da AWS AppSync.

Nota: per ulteriori informazioni sulle autorità di certificazione riconosciute dal AWS AppSync servizio, consulta [Autorità di certificazione \(CA\) riconosciute da AWS AppSync per](#) gli endpoint HTTPS.

Configurazione dei resolver

In questa fase, è possibile connettere l'origine dati http alla query getUser.

Per configurare il resolver:

- Scegli la scheda Schema.
- Nel riquadro Data types (Tipi di dati) a destra sotto il tipo Query, individuare il campo getUser e scegliere Attach (Collega).
- In Data source name (Nome origine dati), scegliere HTTP.
- In Configure the request mapping template (Configura il modello di mappatura della richiesta), incollare il codice seguente:

```
{
  "version": "2018-05-29",
  "method": "GET",
  "params": {
    "headers": {
      "Content-Type": "application/json"
    }
  },
  "resourcePath": $util.toJson("/v1/users/${ctx.args.id}")
}
```

- In Configure the response mapping template (Configura il modello di mappatura della risposta), incollare il codice seguente:

```
## return the body
#if($ctx.result.statusCode == 200)
    ##if response is 200
    $ctx.result.body
#else
    ##if response is not 200, append the response to error block.
    $utils.appendError($ctx.result.body, "$ctx.result.statusCode")
#end
```

- Scegliere la scheda Query quindi eseguire la seguente query:

```
query GetUser{
  getUser(id:1){
    id
    username
  }
}
```

Viene restituita la risposta seguente:

```
{
  "data": {
    "getUser": {
      "id": "1",
      "username": "nadia"
    }
  }
}
```

- Scegli la scheda Schema.
- Nel riquadro Data types (Tipi di dati) a destra sotto Mutation (Mutazione), individuare il campo addUser e scegliere Attach (Collega).
- In Data source name (Nome origine dati), scegliere HTTP.
- In Configure the request mapping template (Configura il modello di mappatura della richiesta), incollare il codice seguente:

```
{
  "version": "2018-05-29",
```

```

    "method": "POST",
    "resourcePath": "/v1/users",
    "params":{
      "headers":{
        "Content-Type": "application/json",
      },
      "body": $util.toJson($ctx.args.userInput)
    }
  }
}

```

- In Configure the response mapping template (Configura il modello di mappatura della risposta), incollare il codice seguente:

```

## Raise a GraphQL field error in case of a datasource invocation error
#if($ctx.error)
  $util.error($ctx.error.message, $ctx.error.type)
#end
## if the response status code is not 200, then return an error. Else return the body
**
#if($ctx.result.statusCode == 200)
  ## If response is 200, return the body.
  $ctx.result.body
#else
  ## If response is not 200, append the response to error block.
  $utils.appendError($ctx.result.body, "$ctx.result.statusCode")
#end

```

- Scegliere la scheda Query quindi eseguire la seguente query:

```

mutation addUser{
  addUser(userInput:{
    id:"2",
    username:"shaggy"
  }){
    id
    username
  }
}

```

Viene restituita la risposta seguente:

```
{
  "data": {
    "getUser": {
      "id": "2",
      "username": "shaggy"
    }
  }
}
```

Richiamo dei servizi AWS

È possibile utilizzare i resolver HTTP per configurare un'interfaccia API GraphQL per i servizi AWS. Le richieste HTTP AWS devono essere firmate con il [processo Signature Version 4](#) in modo da AWS poter identificare chi le ha inviate. AWS AppSync calcola la firma per tuo conto quando associ un ruolo IAM all'origine dati HTTP.

Fornisci due componenti aggiuntivi per richiamare i AWS servizi con resolver HTTP:

- Un ruolo IAM con autorizzazioni per chiamare il servizio AWS APIs
- Configurazione della firma nell'origine dati

Ad esempio, se desideri chiamare l'[ListGraphQLApis operazione](#) con resolver HTTP, devi prima [creare un ruolo IAM](#) che AWS AppSync presuppone la seguente policy allegata:

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "appsync:ListGraphQLApis"
      ],
      "Effect": "Allow",
      "Resource": "*"
    }
  ]
}
```

Quindi, crea l'origine dati HTTP per AWS AppSync. In questo esempio, chiami AWS AppSync nella regione Stati Uniti occidentali (Oregon). Imposta la seguente configurazione HTTP in un file denominato `http.json`, che include la regione di firma e il nome del servizio:

```
{
  "endpoint": "https://appsync.us-west-2.amazonaws.com/",
  "authorizationConfig": {
    "authorizationType": "AWS_IAM",
    "awsIamConfig": {
      "signingRegion": "us-west-2",
      "signingServiceName": "appsync"
    }
  }
}
```

Quindi, utilizzate il AWS CLI per creare l'origine dati con un ruolo associato come segue:

```
aws appsync create-data-source --api-id <API-ID> \
                               --name AWSAppSync \
                               --type HTTP \
                               --http-config file:///http.json \
                               --service-role-arn <ROLE-ARN>
```

Quando colleghi un resolver al campo dello schema, usa il seguente modello di mappatura delle richieste per chiamare: AWS AppSync

```
{
  "version": "2018-05-29",
  "method": "GET",
  "resourcePath": "/v1/apis"
}
```

Quando esegui una query GraphQL per questa fonte di dati, AWS AppSync firma la richiesta utilizzando il ruolo che hai fornito e include la firma nella richiesta. La query restituisce un elenco di AWS AppSync GraphQL APIs nel tuo account in quella AWS regione.

Utilizzo di Aurora Serverless v2 con AWS AppSync

Connect l'API GraphQL ai database Aurora Serverless utilizzando AWS AppSync. Questa integrazione consente di eseguire istruzioni SQL tramite query, mutazioni e sottoscrizioni GraphQL, offrendoti un modo flessibile di interagire con i tuoi dati relazionali.

Note

In questo tutorial viene utilizzata la regione US-EAST-1.

Vantaggi

- Perfetta integrazione tra GraphQL e database relazionali
- Capacità di eseguire operazioni SQL tramite interfacce GraphQL
- Scalabilità senza server con Aurora Serverless v2
- Accesso sicuro ai dati tramite AWS Secrets Manager
- Protezione contro l'SQL injection tramite la sanificazione degli input
- Funzionalità di interrogazione flessibili, tra cui operazioni di filtraggio e intervallo

Casi di utilizzo comune

- Creazione di applicazioni scalabili con requisiti di dati relazionali
- La creazione di APIs ciò richiede sia la flessibilità GraphQL che le funzionalità del database SQL
- Gestione delle operazioni sui dati tramite mutazioni e query GraphQL
- Implementazione di modelli di accesso sicuri al database

In questo tutorial, imparerai quanto segue.

- Configura un cluster Aurora Serverless v2
- Abilita la funzionalità Data API
- Crea e configura strutture di database
- Definizione degli schemi GraphQL per le operazioni del database
- Implementa resolver per query e mutazioni
- Proteggi l'accesso ai dati mediante un'adeguata sanificazione degli input

- Esegui varie operazioni di database tramite interfacce GraphQL

Argomenti

- [Configurazione del cluster di database](#)
- [Abilitazione dell'API di dati](#)
- [Creazione di un database e di una tabella](#)
- [Schema GraphQL](#)
- [Connect la tua API alle operazioni del database](#)
- [Modifica i tuoi dati tramite l'API](#)
- [Recupera i tuoi dati](#)
- [Proteggi l'accesso ai dati](#)

Configurazione del cluster di database

Prima di aggiungere un'origine dati Amazon RDS a AWS AppSync, devi prima abilitare una Data API su un cluster Aurora Serverless v2 e configurare un utilizzo segreto. Gestione dei segreti AWS È possibile creare un cluster Aurora Serverless v2 utilizzando: AWS CLI

```
aws rds create-db-cluster \  
  --db-cluster-identifier appsync-tutorial \  
  --engine aurora-mysql \  
  --engine-version 8.0 \  
  --serverless-v2-scaling-configuration MinCapacity=0,MaxCapacity=1 \  
  --master-username USERNAME \  
  --master-user-password COMPLEX_PASSWORD \  
  --enable-http-endpoint
```

Verrà restituito un ARN per il cluster.

Dopo aver creato il cluster, è necessario aggiungere un'istanza Aurora Serverless v2 utilizzando il comando seguente.

```
aws rds create-db-instance \  
  --db-cluster-identifier appsync-tutorial \  
  --db-instance-identifier appsync-tutorial-instance-1 \  
  --db-instance-class db.serverless \  
  --engine aurora-mysql
```

 Note

L'attivazione di questi endpoint richiede tempo. Puoi verificarne lo stato nella console Amazon RDS nella scheda Connettività e sicurezza del cluster. Puoi anche controllare lo stato del tuo cluster con il seguente AWS CLI comando.

```
aws rds describe-db-clusters \  
  --db-cluster-identifier appsync-tutorial \  
  --query "DBClusters[0].Status"
```

È possibile creare un Secret utilizzando la Gestione dei segreti AWS Console o AWS CLI con un file di input come il seguente utilizzando la USERNAME e COMPLEX_PASSWORD del passaggio precedente.

```
{  
  "username": "USERNAME",  
  "password": "COMPLEX_PASSWORD"  
}
```

Passalo come parametro a AWS CLI:

```
aws secretsmanager create-secret --name HttpRDSSecret --secret-string file://creds.json  
  --region us-east-1
```

Verrà restituito un ARN per il segreto.

Prendi nota dell'ARN del cluster Aurora Serverless e di Secret per utilizzarli successivamente nella AppSync console durante la creazione di un'origine dati.

Abilitazione dell'API di dati

È possibile abilitare l'API di dati sul cluster [seguendo le istruzioni nella documentazione di RDS](#). L'API Data deve essere abilitata prima di aggiungerla come fonte di dati. AppSync

Creazione di un database e di una tabella

Dopo aver abilitato la Data API, puoi assicurarti che funzioni con il `aws rds-data execute-statement` comando contenuto in AWS CLI. Ciò garantirà che il cluster Aurora Serverless sia

configurato correttamente prima di aggiungerlo all'API. AppSync Innanzitutto, creare un database denominato TESTDB con il parametro `--sql`, in questo modo:

```
aws rds-data execute-statement --resource-arn "arn:aws:rds:us-east-1:123456789000:cluster:http-endpoint-test" \  
--schema "mysql" --secret-arn "arn:aws:secretsmanager:us-east-1:123456789000:secret:testHttp2-AmNvc1" \  
--region us-east-1 --sql "create DATABASE TESTDB"
```

Se la creazione viene eseguita senza errori, aggiungere una tabella con il comando `create table`:

```
aws rds-data execute-statement --resource-arn "arn:aws:rds:us-east-1:123456789000:cluster:http-endpoint-test" \  
--schema "mysql" --secret-arn "arn:aws:secretsmanager:us-east-1:123456789000:secret:testHttp2-AmNvc1" \  
--region us-east-1 \  
--sql "create table Pets(id varchar(200), type varchar(200), price float)" --database "TESTDB"
```

Se tutto è stato eseguito senza problemi, puoi passare all'aggiunta del cluster come fonte di dati AppSync nell'API.

Schema GraphQL

Ora che l'API di Aurora Serverless è operativa e dispone di una tabella, creeremo uno schema GraphQL e collegheremo i resolver per l'esecuzione di mutazioni e sottoscrizioni. Crea una nuova API nella AWS AppSync console, vai alla pagina Schema e inserisci quanto segue:

```
type Mutation {  
  createPet(input: CreatePetInput!): Pet  
  updatePet(input: UpdatePetInput!): Pet  
  deletePet(input: DeletePetInput!): Pet  
}  
  
input CreatePetInput {  
  type: PetType  
  price: Float!  
}  
  
input UpdatePetInput {  
  id: ID!
```

```
    type: PetType
    price: Float!
  }

  input DeletePetInput {
    id: ID!
  }

  type Pet {
    id: ID!
    type: PetType
    price: Float
  }

  enum PetType {
    dog
    cat
    fish
    bird
    gecko
  }

  type Query {
    getPet(id: ID!): Pet
    listPets: [Pet]
    listPetsByPriceRange(min: Float, max: Float): [Pet]
  }

  schema {
    query: Query
    mutation: Mutation
  }
```

Salvare lo schema con Save (Salva), accedere alla pagina Data Sources (Origini dati) e creare una nuova origine dati. Selezionare Relational database (Database relazionale) come tipo di origine dati e fornire un nome intellegibile. Utilizzare il nome del database creato nell'ultima fase e l'ARN del cluster in cui tale nome è stato creato. Per il ruolo puoi AppSync creare un nuovo ruolo o crearne uno con una politica simile alla seguente:

JSON

```
{
```

```

    "Version": "2012-10-17",
    "Statement": [
      {
        "Effect": "Allow",
        "Action": [
          "rds-data:BatchExecuteStatement",
          "rds-data:BeginTransaction",
          "rds-data:CommitTransaction",
          "rds-data:ExecuteStatement",
          "rds-data:RollbackTransaction"
        ],
        "Resource": [
          "arn:aws:rds:us-east-1:111122223333:cluster:mydbcluster",
          "arn:aws:rds:us-east-1:111122223333:cluster:mydbcluster:*"
        ]
      },
      {
        "Effect": "Allow",
        "Action": [
          "secretsmanager:GetSecretValue"
        ],
        "Resource": [
          "arn:aws:secretsmanager:us-east-1:111122223333:secret:mysecret",
          "arn:aws:secretsmanager:us-east-1:111122223333:secret:mysecret:*"
        ]
      }
    ]
  }
}

```

Esistono due istruzioni in questa policy a cui viene concesso l'accesso basato sul ruolo. La prima risorsa è il cluster Aurora Serverless e la seconda è l'ARN. Gestione dei segreti AWS Dovrai fornire ENTRAMBI ARNs nella configurazione dell'origine AppSync dati prima di fare clic su Crea.

Passalo come parametro a AWS CLI.

```

aws secretsmanager create-secret \
  --name HttpRDSecret \
  --secret-string file://creds.json \
  --region us-east-1

```

Verrà restituito un ARN per il segreto. Prendi nota dell'ARN del tuo cluster Aurora Serverless e di Secret per dopo quando crei un'origine dati nella console. AWS AppSync

Costruisci la struttura del tuo database

Dopo aver abilitato la tua Data API, puoi assicurarti che funzioni con il `aws rds-data execute-statement` comando contenuto in AWS CLI. Ciò garantirà che il cluster Aurora Serverless v2 sia configurato correttamente prima di aggiungerlo all'API. AWS AppSync Innanzitutto, crea un database chiamato TESTDB con il `--sql` parametro seguente.

```
aws rds-data execute-statement \
    --resource-arn "arn:aws:rds:us-east-1:111122223333:cluster:appsync-
tutorial" \
    --secret-arn "arn:aws:secretsmanager:us-
east-1:111122223333:secret:appsync-tutorial-rds-secret" \
    --region us-east-1 \
    --sql "create DATABASE TESTDB"
```

Se viene eseguito senza errori, aggiungete una tabella con il seguente comando `create table`.

```
aws rds-data execute-statement \
    --resource-arn "arn:aws:rds:us-east-1:111122223333:cluster:http-endpoint-test" \
    --secret-arn "arn:aws:secretsmanager:us-east-1:111122223333:secret:testHttp2-
AmNvc1" \
    --region us-east-1 \
    --sql "create table Pets(id varchar(200), type varchar(200), price float)" \
    --database "TESTDB"
```

Progetta la tua interfaccia API

Dopo che Aurora Serverless v2 Data API è attiva e funzionante con una tabella, crea uno schema GraphQL e collega i resolver per eseguire mutazioni e sottoscrizioni. Crea una nuova API nella console, vai alla pagina Schema nella AWS AppSync console e inserisci quanto segue.

```
type Mutation {
  createPet(input: CreatePetInput!): Pet
  updatePet(input: UpdatePetInput!): Pet
  deletePet(input: DeletePetInput!): Pet
}

input CreatePetInput {
  type: PetType
  price: Float!
```

```
}

input UpdatePetInput {
  id: ID!
  type: PetType
  price: Float!
}

input DeletePetInput {
  id: ID!
}

type Pet {
  id: ID!
  type: PetType
  price: Float
}

enum PetType {
  dog
  cat
  fish
  bird
  gecko
}

type Query {
  getPet(id: ID!): Pet
  listPets: [Pet]
  listPetsByPriceRange(min: Float, max: Float): [Pet]
}

schema {
  query: Query
  mutation: Mutation
}
```

Salvare lo schema con Save (Salva), accedere alla pagina Data Sources (Origini dati) e creare una nuova origine dati. Scegli Database relazionale per il tipo di origine dati e fornisci un nome descrittivo. Utilizzare il nome del database creato nell'ultima fase e l'ARN del cluster in cui tale nome è stato creato. Per il ruolo puoi AWS AppSync creare un nuovo ruolo o crearne uno con una politica simile alla seguente.

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "rds-data:BatchExecuteStatement",
        "rds-data:BeginTransaction",
        "rds-data:CommitTransaction",
        "rds-data:ExecuteStatement",
        "rds-data:RollbackTransaction"
      ],
      "Resource": [
        "arn:aws:rds:us-east-1:111122223333:cluster:mydbcluster",
        "arn:aws:rds:us-east-1:111122223333:cluster:mydbcluster:*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "secretsmanager:GetSecretValue"
      ],
      "Resource": [
        "arn:aws:secretsmanager:us-east-1:111122223333:secret:mysecret",
        "arn:aws:secretsmanager:us-east-1:111122223333:secret:mysecret:*"
      ]
    }
  ]
}
```

Esistono due istruzioni in questa policy a cui viene concesso l'accesso basato sul ruolo. La prima risorsa è il cluster Aurora Serverless v2 e la seconda è l'ARN. Gestione dei segreti AWS Dovrai fornire ENTRAMBI ARNs nella configurazione dell'origine AWS AppSync dati prima di fare clic su Crea.

Connect la tua API alle operazioni del database

Ora che abbiamo uno schema GraphQL valido e un'origine dati RDS, puoi collegare i resolver ai campi GraphQL dello schema. La nostra API offrirà le seguenti funzionalità:

1. crea un animale domestico usando il campo `Mutation.createPet`
2. aggiorna un animale domestico usando il campo `Mutation.updatePet`
3. elimina un animale domestico usando il campo `Mutation.deletePet`
4. ottiene un singolo utilizzo tramite il campo `query.getPET`
5. elenca tutto usando il campo `query.listPets`
6. elenca gli animali domestici in una fascia di prezzo utilizzando il comando `Query.listPetsByPriceRange`

Mutation.createPet

Dall'editor dello schema nella AWS AppSync console, sul lato destro scegli `Attach Resolver for`.
`createPet(input: CreatePetInput!): Pet` Scegliere l'origine dati RDS. Aggiungere il modello seguente nella sezione `request mapping template` (modello di mappatura della richiesta):

```
#set($id=$utils.autoId())
{
  "version": "2018-05-29",
  "statements": [
    "insert into Pets VALUES (:ID, :TYPE, :PRICE)",
    "select * from Pets WHERE id = :ID"
  ],
  "variableMap": {
    ":ID": "$ctx.args.input.id",
    ":TYPE": $util.toJson($ctx.args.input.type),
    ":PRICE": $util.toJson($ctx.args.input.price)
  }
}
```

Il sistema esegue le istruzioni SQL in sequenza, in base all'ordine nell'array delle istruzioni. I risultati verranno restituiti nello stesso ordine. Poiché si tratta di una mutazione, verrà eseguita un'istruzione `select` dopo l'inserimento per recuperare i valori confermati al fine di popolare il modello di mappatura delle risposte GraphQL.

Aggiungere il modello seguente nella sezione `response mapping template` (modello di mappatura della risposta):

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[1][0])
```

Poiché le istruzioni contengono due query SQL, è necessario specificare il secondo risultato nella matrice restituita dal database con: `$utils.rds.toJsonString($ctx.result))[1][0]`.

Mutation.updatePet

Dall'editor dello schema nella AWS AppSync console, scegli Attach Resolver for.

`updatePet(input: UpdatePetInput!): Pet` Scegli la tua fonte di dati RDS. Nella sezione del modello di mappatura della richiesta, aggiungi il seguente modello.

```
{
  "version": "2018-05-29",
  "statements": [
    $util.toJson("update Pets set type=:TYPE, price=:PRICE WHERE id=:ID"),
    $util.toJson("select * from Pets WHERE id = :ID")
  ],
  "variableMap": {
    ":ID": "$ctx.args.input.id",
    ":TYPE": $util.toJson($ctx.args.input.type),
    ":PRICE": $util.toJson($ctx.args.input.price)
  }
}
```

Nella sezione modello di mappatura delle risposte, aggiungi il seguente modello.

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[1][0])
```

Mutation.deletePet

Dall'editor dello schema nella AWS AppSync console, scegli Attach Resolver for.

`deletePet(input: DeletePetInput!): Pet` Scegli la tua fonte di dati RDS. Nella sezione del modello di mappatura della richiesta, aggiungi il seguente modello.

```
{
  "version": "2018-05-29",
  "statements": [
    $util.toJson("select * from Pets WHERE id=:ID"),
    $util.toJson("delete from Pets WHERE id=:ID")
  ],
  "variableMap": {
    ":ID": "$ctx.args.input.id"
  }
}
```

```
}
```

Nella sezione modello di mappatura delle risposte, aggiungi il seguente modello.

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[0][0])
```

Query.getPet

Ora che le mutazioni sono state create per il tuo schema, collega le tre query per mostrare come ottenere singoli elementi, elenchi e applicare il filtro SQL. Dall'editor dello schema nella AWS AppSync console, scegli Attach Resolver for. `getPet(id: ID!): Pet` Scegli la tua fonte di dati RDS. Nella sezione del modello di mappatura della richiesta, aggiungi il seguente modello.

```
{
  "version": "2018-05-29",
  "statements": [
    $util.toJson("select * from Pets WHERE id=:ID")
  ],
  "variableMap": {
    ":ID": "$ctx.args.id"
  }
}
```

Aggiungere il modello seguente nella sezione response mapping template (modello di mappatura della risposta):

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[0][0])
```

Query.listPets

Dall'editor dello schema nella AWS AppSync console, sul lato destro scegli Attach Resolver for. `getPet(id: ID!): Pet` Scegli la tua fonte di dati RDS. Nella sezione del modello di mappatura della richiesta, aggiungi il seguente modello.

```
{
  "version": "2018-05-29",
  "statements": [
    "select * from Pets"
  ]
}
```

```
}
```

Nella sezione modello di mappatura delle risposte, aggiungi il seguente modello.

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[0])
```

Interrogazione. listPetsByPriceRange

Dall'editor dello schema nella AWS AppSync console, sul lato destro scegli Attach Resolver for.

getPet(id: ID!): Pet Scegli la tua fonte di dati RDS. Nella sezione del modello di mappatura della richiesta, aggiungi il seguente modello.

```
{
  "version": "2018-05-29",
  "statements": [
    "select * from Pets where price > :MIN and price < :MAX"
  ],
  "variableMap": {
    ":MAX": $util.toJson($ctx.args.max),
    ":MIN": $util.toJson($ctx.args.min)
  }
}
```

Aggiungere il modello seguente nella sezione response mapping template (modello di mappatura della risposta):

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[0])
```

Modifica i tuoi dati tramite l'API

Ora che sono stati configurati tutti i resolver con le istruzioni SQL e l'API GraphQL è stata collegata all'API di dati per Aurora Serverless, è possibile iniziare a eseguire mutazioni e query. Nella AWS AppSync console, scegli la scheda Query e inserisci quanto segue per creare un animale domestico:

```
mutation add {
  createPet(input : { type:fish, price:10.0 }){
    id
    type
  }
}
```

```
      price
    }
  }
}
```

La risposta dovrebbe contenere l'id, il type (tipo) e il price (prezzo), come indicato di seguito:

```
{
  "data": {
    "createPet": {
      "id": "c6fedbbe-57ad-4da3-860a-ffe8d039882a",
      "type": "fish",
      "price": "10.0"
    }
  }
}
```

È possibile modificare questo elemento eseguendo la mutazione `updatePet`:

```
mutation update {
  updatePet(input : {
    id: ID_PLACEHOLDER,
    type:bird,
    price:50.0
  }){
    id
    type
    price
  }
}
```

Abbiamo utilizzato l'id che è stato restituito dall'operazione `createPet` precedente. Questo sarà un valore univoco per il record poiché il resolver si è basato su `$util.autoId()`. È possibile eliminare un record in modo analogo:

```
mutation delete {
  deletePet(input : {id:ID_PLACEHOLDER}){
    id
    type
    price
  }
}
```

Creare di alcuni record con la prima mutazione con valori diversi per price (prezzo), quindi eseguire alcune query.

Recupera i tuoi dati

Sempre nella scheda Queries della console, usa la seguente istruzione per elencare tutti i record che hai creato.

```
query allpets {  
  listPets {  
    id  
    type  
    price  
  }  
}
```

Sfrutta il predicato SQL WHERE presente *where price > :MIN and price < :MAX* nel nostro modello di mappatura per Query. listPetsByPriceRange con la seguente query GraphQL:

```
query petsByPriceRange {  
  listPetsByPriceRange(min:1, max:11) {  
    id  
    type  
    price  
  }  
}
```

Si dovrebbero visualizzare solo i record con price (prezzo) superiore a \$1 o inferiore a \$10. Infine, è possibile eseguire le query per recuperare singoli record, nel modo seguente:

```
query onePet {  
  getPet(id:ID_PLACEHOLDER){  
    id  
    type  
    price  
  }  
}
```

Proteggi l'accesso ai dati

L'iniezione SQL è una vulnerabilità di sicurezza nelle applicazioni di database. Si verifica quando gli aggressori inseriscono codice SQL dannoso attraverso i campi di input dell'utente. Ciò può consentire l'accesso non autorizzato ai dati del database. Si consiglia di convalidare e disinfettare attentamente tutti gli input dell'utente prima dell'elaborazione, utilizzando `variableMap` per la protezione dagli attacchi di SQL injection. Se non vengono utilizzate mappe variabili, è responsabilità dell'utente sanificare gli argomenti delle loro operazioni GraphQL. Un possibile modo è fornire fasi di convalida specifiche dell'input nel modello di mappatura della richiesta prima dell'esecuzione di un'istruzione SQL sull'API di dati. Vediamo come possiamo modificare il modello di mappatura della richiesta dell'esempio `listPetsByPriceRange`. Anziché basarsi esclusivamente sull'input dell'utente, è possibile procedere nel modo seguente:

```
#set($validMaxPrice = $util.matches("\d{1,3}[,\\.]?(\d{1,2})?", $ctx.args.maxPrice))
#set($validMinPrice = $util.matches("\d{1,3}[,\\.]?(\d{1,2})?", $ctx.args.minPrice))

#if (!$validMaxPrice || !$validMinPrice)
    $util.error("Provided price input is not valid.")
#end
{
    "version": "2018-05-29",
    "statements": [
        "select * from Pets where price > :MIN and price < :MAX"
    ],

    "variableMap": {
        ":MAX": $util.toJson($ctx.args.maxPrice),
        ":MIN": $util.toJson($ctx.args.minPrice)
    }
}
```

Un altro modo per proteggersi da input anomali durante l'esecuzione di resolver sull'API di dati consiste nell'utilizzare istruzioni preparate assieme a procedure memorizzate e input parametrici. Ad esempio, nel resolver per `listPets`, definire la procedura seguente che esegue il `select` come istruzione preparata:

```
CREATE PROCEDURE listPets (IN type_param VARCHAR(200))
BEGIN
```



```

PREPARE stmt FROM 'SELECT * FROM Pets where type=?';
SET @type = type_param;
EXECUTE stmt USING @type;
DEALLOCATE PREPARE stmt;
END

```

Crealo nella tua istanza Aurora Serverless v2.

```

aws rds-data execute-statement --resource-arn "arn:aws:rds:us-
east-1:xxxxxxxxxxxx:cluster:http-endpoint-test" \
--schema "mysql" --secret-arn "arn:aws:secretsmanager:us-
east-1:xxxxxxxxxxxx:secret:httpendpoint-xxxxxx" \
--region us-east-1 --database "DB_NAME" \
--sql "CREATE PROCEDURE listPets (IN type_param VARCHAR(200)) BEGIN PREPARE stmt FROM
'SELECT * FROM Pets where type=?'; SET @type = type_param; EXECUTE stmt USING @type;
DEALLOCATE PREPARE stmt; END"

```

Il codice del resolver risultante per listPets è semplificato poiché ora è sufficiente chiamare la procedura memorizzata. Come minimo, qualsiasi input di stringa deve avere le virgolette singole tra caratteri di [escape](#).

```

#set ($validType = $util.isString($ctx.args.type) && !
$util.isNullOrBlank($ctx.args.type))
#if (!$validType)
    $util.error("Input for 'type' is not valid.", "ValidationError")
#end

{
  "version": "2018-05-29",
  "statements": [
    "CALL listPets(:type)"
  ]
  "variableMap": {
    ":type": $util.toJson($ctx.args.type.replace("'", '''))
  }
}

```

Utilizzo delle stringhe di escape

Usa virgolette singole per contrassegnare l'inizio e la fine delle stringhe letterali in un'istruzione SQL, ad esempio.. 'some string value'. Per consentire l'utilizzo di valori stringa con uno o più caratteri virgolette singole (') all'interno di una stringa, ciascuna virgoletta deve essere sostituita con

due virgolette singole (' '). Ad esempio, se la stringa di input è Nadia 's dog, inserisci il carattere di escape per l'istruzione SQL come segue

```
update Pets set type='Nadia''s dog' WHERE id='1'
```

Utilizzo di risolutori di pipeline in AWS AppSync

Note

Ora supportiamo principalmente il runtime APPSYNC_JS e la relativa documentazione. [Prendi in considerazione l'utilizzo del runtime APPSYNC_JS e delle relative guide qui.](#)

AWS AppSync fornisce un modo semplice per collegare un campo GraphQL a una singola fonte di dati tramite resolver di unità. Tuttavia, un'unica operazione potrebbe non essere sufficiente. I resolver della pipeline offrono la possibilità di eseguire operazioni seriali sulle origini dati. Creazione di funzioni nell'API e collegamento a un resolver della pipeline. Il risultato di ogni funzione eseguita viene reindirizzato alla funzione successiva finché non rimangono altre funzioni da eseguire. Con i resolver della pipeline, è ora possibile creare flussi di lavoro più complessi direttamente in AWS AppSync. In questo tutorial, verrà creata una semplice applicazione per la visualizzazione di immagini, in cui gli utenti possono pubblicare immagini e visualizzare quelle pubblicate dai loro amici.

Impostazione One-Click

Se desideri configurare automaticamente l'endpoint GraphQL AWS AppSync con tutti i resolver configurati e le AWS risorse necessarie, puoi utilizzare il seguente modello: AWS CloudFormation

**Launch Stack**

Con questo stack, nell'account vengono create le seguenti risorse:

- IAM Role per accedere AWS AppSync alle risorse del tuo account
- 2 tabelle DynamoDB
- 1 pool di utenti di Amazon Cognito
- 2 gruppi di pool di utenti di Amazon Cognito
- 3 utenti di pool di utenti di Amazon Cognito

- 1 API AWS AppSync

Al termine del processo di creazione dello AWS CloudFormation stack, riceverai un'e-mail per ciascuno dei tre utenti Amazon Cognito creati. Ogni e-mail contiene una password temporanea che usi per accedere come utente Amazon Cognito alla AWS AppSync console. Salvare le password per la parte restante del tutorial.

Configurazione manuale

Se preferisci eseguire manualmente una step-by-step procedura tramite la AWS AppSync console, segui la procedura di configurazione riportata di seguito.

Configurazione AWS AppSync delle risorse non aziendali

L'API comunica con due tabelle DynamoDB: una tabella di immagini che memorizza le immagini e una tabella di amici che memorizza le relazioni tra gli utenti. L'API è configurata per utilizzare un pool di utenti di Amazon Cognito come tipo di autenticazione. Lo CloudFormation stack seguente configura queste risorse nell'account.



Al termine del processo di creazione dello AWS CloudFormation stack, riceverai un'e-mail per ciascuno dei tre utenti Amazon Cognito creati. Ogni e-mail conterrà una password temporanea utilizzabile per effettuare l'accesso come utente Amazon Cognito alla console di AWS AppSync. Salvare le password per la parte restante del tutorial.

Creazione dell'API GraphQL

Per creare l'API GraphQL in: AWS AppSync

1. Apri la AWS AppSync console e scegli Build From Scratch e scegli Start.
2. Impostare il nome dell'API su AppSyncTutorial-PicturesViewer.
3. Scegli Create (Crea).

La AWS AppSync console crea una nuova API GraphQL utilizzando la modalità di autenticazione della chiave API. Puoi usare la console per configurare ulteriormente l'API GraphQL ed eseguire query sull'API per le parti restanti di questo tutorial.

Configurazione dell'API per GraphQL

Devi configurare l' AWS AppSync API con il pool di utenti Amazon Cognito che hai appena creato.

1. Seleziona la scheda Impostazioni.
2. Nella sezione Authorization Type (Tipo di autorizzazione), scegliere Amazon Cognito User Pool (Pool di utenti di Amazon Cognito).
3. In Configurazione del pool di utenti, scegli US-WEST-2 per la regione.AWS
4. Scegli il pool di UserPool utenti AppSyncTutorial-.
5. Scegliere DENY (NEGA) come Default Action (Operazione predefinita).
6. Lascia vuoto il campo regex del AppId client.
7. Scegli Save (Salva).

L'API è ora configurata per utilizzare un pool di utenti di Amazon Cognito come tipo di autorizzazione.

Configurazione delle sorgenti dati per le tabelle DynamoDB

Dopo aver creato le tabelle DynamoDB, accedi all'API AWS AppSync GraphQL nella console e scegli la scheda Data Sources. Ora creerai un'origine dati AWS AppSync per ciascuna delle tabelle DynamoDB che hai appena creato.

1. Passa alla scheda Data source (Origine dati).
2. Scegli New (Nuovo) per creare una nuova origine dati.
3. Per il nome dell'origine dati, immetti PicturesDynamoDBTable.
4. Scegli Amazon DynamoDB table (Tabella Amazon DynamoDB) come tipo di origine dati.
5. Scegli US-WEST-2 per la regione.
6. Dall'elenco delle tabelle, scegli la tabella AppSyncTutorial-Pictures DynamoDB.
7. Scegliere Existing role (Ruolo esistente) nella sezione Create or use an existing role (Crea o usa un ruolo esistente).
8. Scegli il ruolo appena creato dal modello. CloudFormation Se non hai cambiato il ResourceNamePrefix, il nome del ruolo dovrebbe essere AppSyncTutorial-Dynamo DBRole.
9. Scegli Create (Crea).

Ripeti la stessa procedura per la tabella degli amici, il nome della tabella DynamoDB dovrebbe essere `-Friends` se non hai modificato `ResourceNamePrefix` il parametro al momento della creazione dello stack. CloudFormation

Creazione dello schema GraphQL

Ora che le sorgenti dati sono collegate alle tabelle DynamoDB, creiamo uno schema GraphQL. Dall'editor di schemi nella AWS AppSync console, assicurati che lo schema corrisponda al seguente schema:

```
schema {
  query: Query
  mutation: Mutation
}

type Mutation {
  createPicture(input: CreatePictureInput!): Picture!
  @aws_auth(cognito_groups: ["Admins"])
  createFriendship(id: ID!, target: ID!): Boolean
  @aws_auth(cognito_groups: ["Admins"])
}

type Query {
  getPicturesByOwner(id: ID!): [Picture]
  @aws_auth(cognito_groups: ["Admins", "Viewers"])
}

type Picture {
  id: ID!
  owner: ID!
  src: String
}

input CreatePictureInput {
  owner: ID!
  src: String!
}
```

Scegliere **Save Schema** (Salva schema) per salvare lo schema.

Alcuni campi dello schema sono stati annotati con la direttiva `@aws_auth`. Poiché l'operazione predefinita dell'API è impostata su DENY (NEGA), l'API rifiuta tutti gli utenti che non sono membri

dei gruppi inclusi nella direttiva `@aws_auth`. Per ulteriori informazioni su come proteggere l'API, è possibile leggere la pagina sulla [sicurezza](#). In questo caso, solo gli utenti amministratori hanno accesso ai campi `Mutation.createPicture` e `Mutation.createFriendship`, mentre gli utenti membri dei gruppi Admins o Viewers possono accedere alla Query `getPicturesByCampo Proprietario`. Tutti gli altri utenti non hanno accesso.

Configurazione dei resolver

Ora che si dispone di un valido schema GraphQL e di due origini dati, è possibile collegare i resolver ai campi GraphQL dello schema. L'API offre le seguenti funzionalità:

- Creazione di un'immagine tramite il campo `Mutation.createPicture`
- Creazione di una richiesta di amicizia tramite il campo `Mutation.createFriendship`
- Recupero di un'immagine tramite il campo `Query.getPicture`

`Mutation.createPicture`

Dall'editor dello schema nella AWS AppSync console, sul lato destro scegli **Attach Resolver for**. `createPicture(input: CreatePictureInput!): Picture!` Scegli l'origine dati `PicturesDynamoDBTableDynamoDB`. Aggiungere il modello seguente nella sezione **request mapping template** (modello di mappatura della richiesta):

```
#set($id = $util.autoId())

{
  "version" : "2018-05-29",
  "operation" : "PutItem",
  "key" : {
    "id" : $util.dynamodb.toDynamoDBJson($id),
    "owner": $util.dynamodb.toDynamoDBJson($ctx.args.input.owner)
  },
  "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args.input)
}
```

Aggiungere il modello seguente nella sezione **response mapping template** (modello di mappatura della risposta):

```
#if($ctx.error)
    $util.error($ctx.error.message, $ctx.error.type)
#end
$util.toJson($ctx.result)
```

La funzionalità per la creazione di immagini è stata creata. Verrà salvata un'immagine nella tabella `pictures` con un UUID casuale come id dell'immagine e il nome utente di Cognito come proprietario dell'immagine.

Mutation.createFriendship

Dall'editor dello schema nella AWS AppSync console, sul lato destro scegli `Attach Resolver` for `createFriendship(id: ID!, target: ID!): Boolean`. Scegli l'origine dati `FriendsDynamoDBTableDynamoDB`. Aggiungere il modello seguente nella sezione `request mapping template` (modello di mappatura della richiesta):

```
#set($userToFriendFriendship = { "userId" : "$ctx.args.id", "friendId":
  "$ctx.args.target" })
#set($friendToUserFriendship = { "userId" : "$ctx.args.target", "friendId":
  "$ctx.args.id" })
#set($friendsItems = [$util.dynamodb.toMapValues($userToFriendFriendship),
  $util.dynamodb.toMapValues($friendToUserFriendship)])

{
  "version" : "2018-05-29",
  "operation" : "BatchPutItem",
  "tables" : {
    ## Replace 'AppSyncTutorial-' default below with the ResourceNamePrefix you
    provided in the CloudFormation template
    "AppSyncTutorial-Friends": $util.toJson($friendsItems)
  }
}
```

Importante: nel modello di `BatchPutItem` richiesta, deve essere presente il nome esatto della tabella `DynamoDB`. Il nome di tabella predefinito è `AppSyncTutorial -Friends`. Se si utilizza il nome di tabella sbagliato, viene visualizzato un errore quando si AppSync tenta di assumere il ruolo fornito.

Per semplicità, in questo tutorial, procedi come se la richiesta di amicizia fosse stata approvata e salva la voce della relazione direttamente nella `AppSyncTutorialFriendstabella`.

Di fatto, si stanno memorizzando due voci per ogni richiesta di amicizia dato che la relazione è bidirezionale. Per ulteriori dettagli sulle best practice di Amazon DynamoDB per many-to-many rappresentare le relazioni, consulta [DynamoDB Best Practices](#).

Aggiungere il modello seguente nella sezione response mapping template (modello di mappatura della risposta):

```
#if($ctx.error)
    $util.error($ctx.error.message, $ctx.error.type)
#end
true
```

Nota: verificare che il proprio modello di richiesta contenga il nome corretto della tabella. Il nome predefinito è AppSyncTutorial-Friends, ma il nome della tabella potrebbe essere diverso se modifichi il parametro. CloudFormation ResourceNamePrefix

Interrogazione. getPicturesByProprietario

Ora che si dispone di richieste di amicizia e immagini, è necessario fornire la possibilità agli utenti di visualizzare le immagini dei loro amici. Per soddisfare questo requisito, occorre prima verificare che il richiedente sia amico del proprietario, quindi è necessario eseguire una query per ottenere l'immagine.

Poiché questa funzionalità richiede due operazioni sull'origine dati, si creano due funzioni. La prima funzione, isFriend, verifica se il richiedente e il proprietario sono amici. La seconda funzione, getPicturesByOwner, recupera le immagini richieste con un ID proprietario. Diamo un'occhiata al flusso di esecuzione riportato di seguito per il resolver proposto sulla Query. getPicturesByCampo proprietario:

1. Modello di mappatura Before: prepara il contesto e gli argomenti di input per il campo.
2. Funzione isFriend: verifica se il richiedente è il proprietario dell'immagine. In caso contrario, verifica se gli utenti richiedenti e proprietari sono amici eseguendo un'operazione GetItem DynamoDB sulla tabella degli amici.
3. getPicturesByFunzione Owner: recupera le immagini dalla tabella Pictures utilizzando un'operazione di interrogazione DynamoDB sull'indice secondario globale owner-index.
4. Modello di mappatura After: mappa le immagini risultanti in modo che gli attributi di DynamoDB siano mappati correttamente con i campi dei tipi GraphQL previsti.

Innanzitutto, verranno create le funzioni.

Funzione isFriend

1. Selezionare la scheda Functions (Funzioni).
2. Scegliere Create Function (Crea funzione) per creare una funzione.
3. Per il nome dell'origine dati, immetti FriendsDynamoDBTable.
4. Come nome della funzione, immettere isFriend.
5. Nell'area di testo del modello di mappatura della richiesta, incollare il seguente modello:

```
#set($ownerId = $ctx.prev.result.owner)
#set($callerId = $ctx.prev.result.callerId)

## if the owner is the caller, no need to make the check
#if($ownerId == $callerId)
    #return($ctx.prev.result)
#end

{
    "version" : "2018-05-29",

    "operation" : "GetItem",

    "key" : {
        "userId" : $util.dynamodb.toDynamoDBJson($callerId),
        "friendId" : $util.dynamodb.toDynamoDBJson($ownerId)
    }
}
```

6. Nell'area di testo del modello di mappatura della risposta, incollare il seguente modello:

```
#if($ctx.error)
    $util.error("Unable to retrieve friend mapping message: ${ctx.error.message}",
    $ctx.error.type)
#end

## if the users aren't friends
#if(!$ctx.result)
    $util.unauthorized()
#end

$util.toJson($ctx.prev.result)
```

7. Selezionare Create function (Crea funzione).

Risultato: è stata creata la funzione isFriend.

getPicturesByFunzione del proprietario

1. Selezionare la scheda Functions (Funzioni).
2. Scegliere Create Function (Crea funzione) per creare una funzione.
3. Per il nome dell'origine dati, immetti PicturesDynamoDBTable.
4. Come nome della funzione, immettere getPicturesByOwner.
5. Nell'area di testo del modello di mappatura della richiesta, incollare il seguente modello:

```
{
  "version" : "2018-05-29",
  "operation" : "Query",
  "query" : {
    "expression": "#owner = :owner",
    "expressionNames": {
      "#owner" : "owner"
    },
    "expressionValues" : {
      ":owner" : $util.dynamodb.toDynamoDBJson($ctx.prev.result.owner)
    }
  },
  "index": "owner-index"
}
```

6. Nell'area di testo del modello di mappatura della risposta, incollare il seguente modello:

```
#if($ctx.error)
  $util.error($ctx.error.message, $ctx.error.type)
#end

$util.toJson($ctx.result)
```

7. Selezionare Create function (Crea funzione).

Risultato: hai creato la funzione `getPicturesByOwner`. Ora che le funzioni sono state create, collega un resolver di pipeline alla Query `getPicturesByCampo Proprietario`.

Dall'editor dello schema nella AWS AppSync console, sul lato destro scegli **Attach Resolver for**. `Query.getPicturesByOwner(id: ID!): [Picture]` Nella pagina seguente, selezionare il collegamento **Convert to pipeline resolver** (Converti in resolver della pipeline) che viene visualizzato sotto l'origine dati nell'elenco a discesa. Utilizzare la procedura seguente per il modello di mappatura **Before**:

```
#set($result = { "owner": $ctx.args.id, "callerId": $ctx.identity.username })
$util.toJson($result)
```

Nella sezione **after mapping template** (modello di mappatura **After**), utilizzare la procedura seguente:

```
#foreach($picture in $ctx.result.items)
  ## prepend "src://" to picture.src property
  #set($picture['src'] = "src://${picture['src']}")
#end
$util.toJson($ctx.result.items)
```

Scegliere **Create Resolver** (Crea resolver). Il primo resolver della pipeline è stato ora collegato. Nella stessa pagina, aggiungere le due funzioni create in precedenza. Nella sezione delle funzioni, scegliere **Add A Function** (Aggiungi funzione), quindi scegliere o digitare il nome della prima funzione, `isFriend`. Aggiungi la seconda funzione seguendo la stessa procedura per la funzione `getPicturesByOwner`. Assicurati che la funzione `isFriend` compaia per prima nell'elenco seguita dalla funzione `getPicturesByOwner`. È possibile utilizzare le frecce **Su** e **Giù** per riorganizzare le funzioni per ordinarne l'esecuzione nella pipeline.

Una volta creato il resolver della pipeline e collegate le funzioni, è opportuno testare la nuova API GraphQL.

Test dell'API GraphQL

Innanzitutto, occorre inserire immagini e richieste di amicizia eseguendo alcune mutazioni con l'utente con privilegi di amministratore creato. Sul lato sinistro della AWS AppSync console, scegli la scheda **Interrogazioni**.

Mutazione `createPicture`

1. Nella AWS AppSync console, scegli la scheda **Interrogazioni**.

2. Scegliere Login With User Pools (Accedi con pool di utenti).
3. Nella modalità modale, inserisci l'ID client Cognito Sample creato dallo stack (ad esempio CloudFormation 37solo6mmhh7k4v63cqdfgdg5d).
4. Immettete il nome utente CloudFormation che avete passato come parametro allo stack. L'impostazione predefinita è nadia.
5. Utilizza la password temporanea che è stata inviata all'e-mail che hai fornito come parametro per lo CloudFormation stack (ad esempio, UserPoolUserEmail).
6. Selezionare Login (Accesso). Ora dovresti vedere il pulsante rinominato Logout nadia, o qualunque nome utente tu abbia scelto durante la creazione dello CloudFormation stack (ovvero,). UserPoolUsername

A questo punto trasmetteremo alcune mutazioni createPicture per popolare la tabella delle immagini. Eseguire la seguente query GraphQL nella console:

```
mutation {  
  createPicture(input:{  
    owner: "nadia"  
    src: "nadia.jpg"  
  }) {  
    id  
    owner  
    src  
  }  
}
```

La risposta dovrebbe essere simile alla seguente:

```
{  
  "data": {  
    "createPicture": {  
      "id": "c6fedbbe-57ad-4da3-860a-ffe8d039882a",  
      "owner": "nadia",  
      "src": "nadia.jpg"  
    }  
  }  
}
```

Aggiungiamo alcune altre immagini:

```
mutation {  
  createPicture(input:{  
    owner: "shaggy"  
    src: "shaggy.jpg"  
  }) {  
    id  
    owner  
    src  
  }  
}
```

```
mutation {  
  createPicture(input:{  
    owner: "rex"  
    src: "rex.jpg"  
  }) {  
    id  
    owner  
    src  
  }  
}
```

Sono state aggiunte tre immagini utilizzando nadia come utente con privilegi di amministratore.

Mutazione createFriendship

Ora aggiungiamo una richiesta di amicizia. Eseguire le seguenti mutazioni nella console.

Nota: è necessario essere connessi con l'utente con privilegi di amministratore (l'utente predefinito è nadia).

```
mutation {  
  createFriendship(id: "nadia", target: "shaggy")  
}
```

La risposta dovrebbe essere simile alla seguente:

```
{  
  "data": {  
    "createFriendship": true  
  }  
}
```

```
}
```

nadia e shaggy sono amici. rex non è amico di altri utenti.

getPicturesByRichiesta del proprietario

Per questa fase, effettuare l'accesso con l'utente nadia, utilizzando pool di utenti di Cognito e le credenziali impostate all'inizio di questo tutorial. Come utente nadia, recuperare le immagini di proprietà di shaggy.

```
query {  
  getPicturesByOwner(id: "shaggy") {  
    id  
    owner  
    src  
  }  
}
```

Poiché nadia e shaggy sono amici, la query dovrebbe restituire l'immagine corrispondente.

```
{  
  "data": {  
    "getPicturesByOwner": [  
      {  
        "id": "05a16fba-cc29-41ee-a8d5-4e791f4f1079",  
        "owner": "shaggy",  
        "src": "src://shaggy.jpg"  
      }  
    ]  
  }  
}
```

Analogamente, anche un eventuale tentativo di nadia di recuperare proprie immagini andrebbe a buon fine. Il pipeline resolver è stato ottimizzato per evitare l'esecuzione dell'operazione `GetItem` `isFriend` in questo caso. Provare la seguente query:

```
query {  
  getPicturesByOwner(id: "nadia") {  
    id  
    owner  
    src  
  }  
}
```

```
}
}
```

Se si attiva la registrazione per le API (nel riquadro Settings (Impostazioni)), impostare il livello di debug a ALL ed eseguire di nuovo la stessa query; verranno restituiti i log per l'esecuzione effettiva. Esaminando i log, è possibile stabilire se la funzione `isFriend` ha restituito un output nelle prime fasi del modello di mappatura della richiesta:

```
{
  "errors": [],
  "mappingTemplateType": "Request Mapping",
  "path": "[getPicturesByOwner]",
  "resolverArn": "arn:aws:appsync:us-west-2:XXXX:apis/XXXX/types/Query/fields/
getPicturesByOwner",
  "functionArn": "arn:aws:appsync:us-west-2:XXXX:apis/XXXX/functions/
o2f42p2jrfdl3dw7s6xub2csdfs",
  "functionName": "isFriend",
  "earlyReturnedValue": {
    "owner": "nadia",
    "callerId": "nadia"
  },
  "context": {
    "arguments": {
      "id": "nadia"
    },
    "prev": {
      "result": {
        "owner": "nadia",
        "callerId": "nadia"
      }
    },
    "stash": {},
    "outErrors": []
  },
  "fieldInError": false
}
```

La `earlyReturnedValue` chiave rappresenta i dati restituiti dalla direttiva `#return`.

Infine, anche se `rex` è membro del Viewers Cognito UserPool Group e poiché `rex` non è amico di nessuno, non potrà accedere a nessuna delle immagini di proprietà di Shaggy o Nadia. Se si effettua l'accesso come `rex` nella console e si esegue la query seguente:

```
query {  
  getPicturesByOwner(id: "nadia") {  
    id  
    owner  
    src  
  }  
}
```

Si otterrà il seguente errore per mancanza di autorizzazioni:

```
{  
  "data": {  
    "getPicturesByOwner": null  
  },  
  "errors": [  
    {  
      "path": [  
        "getPicturesByOwner"  
      ],  
      "data": null,  
      "errorType": "Unauthorized",  
      "errorInfo": null,  
      "locations": [  
        {  
          "line": 2,  
          "column": 9,  
          "sourceName": null  
        }  
      ],  
      "message": "Not Authorized to access getPicturesByOwner on type Query"  
    }  
  ]  
}
```

L'autorizzazione complessa è stata correttamente implementata utilizzando i resolver della pipeline.

Utilizzo delle operazioni Delta Sync su sorgenti dati con versione in AWS AppSync

Note

Ora supportiamo principalmente il runtime APPSYNC_JS e la relativa documentazione. [Prendi in considerazione l'utilizzo del runtime APPSYNC_JS e delle relative guide qui.](#)

Le applicazioni client AWS AppSync archiviano i dati memorizzando nella cache le risposte GraphQL localmente sul disco di un'applicazione. mobile/web Le origini dati e le operazioni Sync con versioni offrono ai clienti la possibilità di eseguire il processo di sincronizzazione utilizzando un unico resolver. In tal modo, per i client è possibile rigenerare la cache locale con i risultati della query di base, dalla notevole quantità di record, per poi acquisire solo i dati variati dall'ultima query (gli aggiornamenti differenziali). Consentendo ai client di eseguire la rigenerazione di base della cache con una richiesta iniziale e con aggiornamenti incrementali con una seconda richiesta, è possibile trasferire le risorse e le operazioni di calcolo dall'applicazione client al back-end. Ciò è sostanzialmente più efficiente per le applicazioni client che spesso passano da uno stato online a uno stato offline.

Per implementare Delta Sync, la query Sync utilizza l'operazione Sync su un'origine dati con versione. Quando una AWS AppSync mutazione modifica un elemento in un'origine dati con versione, un record di tale modifica verrà archiviato anche nella tabella Delta. Puoi scegliere di utilizzare diverse tabelle Delta (ad esempio una per tipo, una per area di dominio) per altre fonti di dati con versione o una singola tabella Delta per la tua API. AWS AppSync sconsiglia di utilizzare una singola tabella Delta per più tabelle per APIs evitare la collisione delle chiavi primarie.

Inoltre, i client che si avvalgono della sincronizzazione differenziale possono ricevere una sottoscrizione sotto forma di argomento, per poi coordinarne le operazioni di scrittura e riconnessione nelle transizioni tra offline e online. A tale scopo, la sincronizzazione automatica riprende automaticamente le sottoscrizioni, inclusa una strategia di "backoff esponenziale e riprova con jitter" su più scenari di errore di rete e archivia gli eventi in una coda. Esegue poi, prima dell'unione degli eventi in coda, la query di base o differenziale appropriata e, infine, elabora le sottoscrizioni normalmente.

[La documentazione per le opzioni di configurazione del client, incluso Amplify, è disponibile sul sito Web di DataStore Amplify Framework.](#) Questa documentazione illustra come impostare origini dati

e operazioni Sync DynamoDB con versioni per lavorare con il client Delta Sync per un accesso ottimale ai dati.

Impostazione One-Click

Per configurare automaticamente l'endpoint GraphQL AWS AppSync con tutti i resolver configurati e le risorse necessarie AWS , usa questo modello: AWS CloudFormation



Con questo stack, nell'account vengono create le seguenti risorse:

- 2 tabelle DynamoDB (Base e Delta)
- 1 AWS AppSync API con chiave API
- 1 ruolo IAM con policy per le tabelle DynamoDB

Due tabelle tornano utili per partizionare le query di sincronizzazione differenziali in una seconda tabella che funge da registro degli eventi non rilevati con i client offline. Per mantenere efficienti le query sulla tabella delta, [TTLsAmazon](#) DynamoDB viene utilizzato per organizzare automaticamente gli eventi in base alle necessità. L'ora TTL è configurabile per le tue esigenze sull'origine dati (puoi impostarla come 1 ora, 1 giorno ecc.).

Schema

Per dimostrare Delta Sync, l'applicazione di esempio crea uno schema Posts supportato da una tabella Base e Delta in DynamoDB. AWS AppSync scrive automaticamente le mutazioni in entrambe le tabelle. Le query di sincronizzazione estraggono i record dalla tabella Base o Delta, in base alle esigenze, e viene definita un'unica sottoscrizione che mostra in che modo i client possono avvalersene nella propria logica di riconnessione.

```
input CreatePostInput {  
  author: String!  
  title: String!  
  content: String!  
  url: String  
  ups: Int  
  downs: Int  
  _version: Int  
}
```

```
interface Connection {
  nextToken: String
  startedAt: AWSTimestamp!
}

type Mutation {
  createPost(input: CreatePostInput!): Post
  updatePost(input: UpdatePostInput!): Post
  deletePost(input: DeletePostInput!): Post
}

type Post {
  id: ID!
  author: String!
  title: String!
  content: String!
  url: AWSURL
  ups: Int
  downs: Int
  _version: Int
  _deleted: Boolean
  _lastChangedAt: AWSTimestamp!
}

type PostConnection implements Connection {
  items: [Post!]!
  nextToken: String
  startedAt: AWSTimestamp!
}

type Query {
  getPost(id: ID!): Post
  syncPosts(limit: Int, nextToken: String, lastSync: AWSTimestamp): PostConnection!
}

type Subscription {
  onCreatePost: Post
    @aws_subscribe(mutations: ["createPost"])
  onUpdatePost: Post
    @aws_subscribe(mutations: ["updatePost"])
  onDeletePost: Post
    @aws_subscribe(mutations: ["deletePost"])
}
```

```
input DeletePostInput {
  id: ID!
  _version: Int!
}

input UpdatePostInput {
  id: ID!
  author: String
  title: String
  content: String
  url: String
  ups: Int
  downs: Int
  _version: Int!
}

schema {
  query: Query
  mutation: Mutation
  subscription: Subscription
}
```

Lo schema di GraphQL è standard, tuttavia conviene soffermarsi su un paio di aspetti, prima di procedere. Innanzitutto, tutte le mutazioni scrivono automaticamente prima nella tabella Base e poi nella tabella Delta. La tabella Base è la fonte centrale di attendibilità per lo stato, mentre la Differenziale funge da registro. Se non si passa in `lastSync: AWSTimestamp`, la query `syncPosts` viene eseguita sulla tabella Base per rigenerare la cache e, periodicamente, come procedura di aggiornamento globale, per i casi limite in cui i client restano offline più a lungo rispetto alla tempistica di TTL configurata nella tabella Delta. Se si passa in `lastSync: AWSTimestamp`, la query `syncPosts` viene eseguita sulla tabella Delta e viene utilizzata dai client per recuperare gli eventi cambiati dall'ultimo passaggio offline. L'amplificazione dei client passa automaticamente il valore `lastSync: AWSTimestamp` e persiste sul disco in modo appropriato.

Il campo `_deleted` su `Post` viene utilizzato per operazioni DELETE. Quando i client sono offline e i record vengono rimossi dalla tabella Base, questo attributo notifica ai client l'eliminazione delle voci dalla cache locale in fase di sincronizzazione. Se i client restano offline per un periodo prolungato e la voce viene rimossa prima che si possa recuperare questo valore con una query di sincronizzazione differenziale, l'evento di aggiornamento globale nella query di base (configurabile nel client) viene eseguito e rimuove la voce dalla cache. Questo campo viene contrassegnato come facoltativo

perché restituisce un valore solo all'esecuzione di una query di sincronizzazione responsabile dell'eliminazione di elementi.

Mutazioni

Per tutte le mutazioni, AWS AppSync esegue un'Create/Update/Deleteoperazione standard nella tabella Base e registra automaticamente anche la modifica nella tabella Delta. È possibile ridurre o estendere il tempo di permanenza dei record, modificando il valore `DeltaSyncTableTTL` nell'origine dati. Alle organizzazioni che acquisiscono ed elaborano dati frequentemente e rapidamente si addice un periodo di permanenza ridotto. Se invece i client restano offline a lungo, è consigliabile impostare un periodo di permanenza prolungato.

Query di sincronizzazione

La query di base è un'operazione di sincronizzazione DynamoDB senza `lastSync` un valore specificato. Lo standard torna utile a quelle organizzazioni che eseguono la query di base solo all'avvio e, successivamente, a cadenza periodica.

La delta query è un'operazione di sincronizzazione DynamoDB con `lastSync` un valore specificato. La query delta viene eseguita ogni volta che il client torna online da uno stato offline (se le tempistiche per le attività periodiche della query di base non hanno avviato l'esecuzione). I client monitorano automaticamente l'ultima volta che hanno eseguito correttamente una query per sincronizzare i dati.

Quando viene eseguita una query delta, il resolver della query utilizza `ds_pk` e `ds_sk` per eseguire query solo per i record che sono stati modificati dall'ultima volta che il client ha eseguito una sincronizzazione. Il client memorizza nella cache la risposta di GraphQL appropriata.

Per ulteriori informazioni sull'esecuzione di query di sincronizzazione, vedere la [documentazione relativa all'operazione di sincronizzazione](#).

Esempio

Iniziamo prima chiamando una mutazione `createPost` per creare un elemento:

```
mutation create {  
  createPost(input: {author: "Nadia", title: "My First Post", content: "Hello World"})  
  {  
    id
```

```
    author
    title
    content
    _version
    _lastChangedAt
    _deleted
  }
}
```

Il valore di restituzione di questa mutazione sarà il seguente:

```
{
  "data": {
    "createPost": {
      "id": "81d36bbb-1579-4efe-92b8-2e3f679f628b",
      "author": "Nadia",
      "title": "My First Post",
      "content": "Hello World",
      "_version": 1,
      "_lastChangedAt": 1574469356331,
      "_deleted": null
    }
  }
}
```

Se si esamina il contenuto della tabella Base si vedrà un record simile a:

```
{
  "_lastChangedAt": {
    "N": "1574469356331"
  },
  "_version": {
    "N": "1"
  },
  "author": {
    "S": "Nadia"
  },
  "content": {
    "S": "Hello World"
  },
  "id": {
    "S": "81d36bbb-1579-4efe-92b8-2e3f679f628b"
  },
}
```

```
"title": {  
  "S": "My First Post"  
}  
}
```

Se si esamina il contenuto della tabella Delta si vedrà un record simile a:

```
{  
  "_lastChangedAt": {  
    "N": "1574469356331"  
  },  
  "_ttl": {  
    "N": "1574472956"  
  },  
  "_version": {  
    "N": "1"  
  },  
  "author": {  
    "S": "Nadia"  
  },  
  "content": {  
    "S": "Hello World"  
  },  
  "ds_pk": {  
    "S": "AppSync-delta-sync-post:2019-11-23"  
  },  
  "ds_sk": {  
    "S": "00:35:56.331:81d36bbb-1579-4efe-92b8-2e3f679f628b:1"  
  },  
  "id": {  
    "S": "81d36bbb-1579-4efe-92b8-2e3f679f628b"  
  },  
  "title": {  
    "S": "My First Post"  
  }  
}
```

Ora possiamo simulare una query di base che un client eseguirà per rigenerare il suo archivio dati locale utilizzando una query `syncPosts` come:

```
query baseQuery {  
  syncPosts(limit: 100, lastSync: null, nextToken: null) {  
    items {
```

```
    id
    author
    title
    content
    _version
    _lastChangedAt
  }
  startedAt
  nextToken
}
```

Il valore restituito di questa query di base sarà il seguente:

```
{
  "data": {
    "syncPosts": {
      "items": [
        {
          "id": "81d36bbb-1579-4efe-92b8-2e3f679f628b",
          "author": "Nadia",
          "title": "My First Post",
          "content": "Hello World",
          "_version": 1,
          "_lastChangedAt": 1574469356331
        }
      ],
      "startedAt": 1574469602238,
      "nextToken": null
    }
  }
}
```

Salveremo il valore `startedAt` in seguito per simulare una query Delta, ma prima dobbiamo apportare una modifica alla nostra tabella. Usiamo la mutazione `updatePost` per modificare il nostro Post esistente:

```
mutation updatePost {
  updatePost(input: {id: "81d36bbb-1579-4efe-92b8-2e3f679f628b", _version: 1, title:
    "Actually this is my Second Post"}) {
    id
    author
```



```
    title
    content
    _version
    _lastChangedAt
    _deleted
  }
}
```

Il valore di restituzione di questa mutazione sarà il seguente:

```
{
  "data": {
    "updatePost": {
      "id": "81d36bbb-1579-4efe-92b8-2e3f679f628b",
      "author": "Nadia",
      "title": "Actually this is my Second Post",
      "content": "Hello World",
      "_version": 2,
      "_lastChangedAt": 1574469851417,
      "_deleted": null
    }
  }
}
```

Se esamiini il contenuto della tabella Base ora, dovresti vedere l'elemento aggiornato:

```
{
  "_lastChangedAt": {
    "N": "1574469851417"
  },
  "_version": {
    "N": "2"
  },
  "author": {
    "S": "Nadia"
  },
  "content": {
    "S": "Hello World"
  },
  "id": {
    "S": "81d36bbb-1579-4efe-92b8-2e3f679f628b"
  },
  "title": {
```

```
    "S": "Actually this is my Second Post"
  }
}
```

Se esamini il contenuto della tabella Delta ora, dovresti vedere due record:

1. Un record al momento della creazione dell'elemento
2. Un record per quando l'elemento è stato aggiornato.

Il nuovo elemento sarà simile a:

```
{
  "_lastChangedAt": {
    "N": "1574469851417"
  },
  "_ttl": {
    "N": "1574473451"
  },
  "_version": {
    "N": "2"
  },
  "author": {
    "S": "Nadia"
  },
  "content": {
    "S": "Hello World"
  },
  "ds_pk": {
    "S": "AppSync-delta-sync-post:2019-11-23"
  },
  "ds_sk": {
    "S": "00:44:11.417:81d36bbb-1579-4efe-92b8-2e3f679f628b:2"
  },
  "id": {
    "S": "81d36bbb-1579-4efe-92b8-2e3f679f628b"
  },
  "title": {
    "S": "Actually this is my Second Post"
  }
}
```

Ora possiamo simulare una query Delta per recuperare le modifiche che si sono verificate quando il client era offline. Useremo il valore `startedAt` restituito dalla nostra query di base per fare la richiesta:

```
query delta {
  syncPosts(limit: 100, lastSync: 1574469602238, nextToken: null) {
    items {
      id
      author
      title
      content
      _version
    }
    startedAt
    nextToken
  }
}
```

Il valore restituito di questa query Delta sarà il seguente:

```
{
  "data": {
    "syncPosts": {
      "items": [
        {
          "id": "81d36bbb-1579-4efe-92b8-2e3f679f628b",
          "author": "Nadia",
          "title": "Actually this is my Second Post",
          "content": "Hello World",
          "_version": 2
        }
      ],
      "startedAt": 1574470400808,
      "nextToken": null
    }
  }
}
```

Configurazione e impostazioni in AWS AppSync

AWS AppSync ti consente di:

- Memorizza nella cache i dati che vengono richiesti spesso ma che è improbabile che cambino da una richiesta all'altra. Ciò può ridurre il carico sui resolver. Per ulteriori informazioni, consulta [the section called “Configurazione della memorizzazione nella cache lato server e della compressione del payload delle API”](#).
- Oggetti Version GraphQL per gestire ed evitare conflitti tra più client. Per ulteriori informazioni, consulta [the section called “Operazioni di controllo delle versioni, rilevamento dei conflitti e sincronizzazione per DynamoDB”](#).
- Usa nomi di dominio personalizzati per configurare un unico dominio memorabile che funzioni sia per GraphQL che in tempo reale. APIs Per ulteriori informazioni, consulta [Configurazione dei nomi di dominio personalizzati](#).
- Consenti l'accesso al tuo GraphQL APIs tramite un VPC. Per ulteriori informazioni, consulta [Using AWS AppSync Private APIs](#).
- Condividi il tuo GraphQL APIs tramite un'integrazione con AWS Resource Access Manager Per ulteriori informazioni, consulta [Condivisione di AWS AppSync APIs](#).
- Abilita l'introspezione e imposta la profondità delle query e i limiti del resolver per query. [Per ulteriori informazioni, consulta Limiti di configurazione](#).
- Usa le variabili di ambiente per regolare il comportamento dei AWS AppSync resolver e delle funzioni senza aggiornare il codice. Per ulteriori informazioni, consulta [Usare le variabili di ambiente in AWS AppSync](#).

Inoltre, AWS AppSync include i seguenti AWS strumenti standard per la registrazione, il monitoraggio e la tracciabilità:

- [Login AWS CloudTrail](#)
- [Monitoraggio con Amazon CloudWatch](#)
- [Tracciamento con AWS X-Ray](#)

Configurazione della memorizzazione nella cache lato server e della compressione del payload delle API in AWS AppSync

AWS AppSync le funzionalità di caching dei dati lato server rendono disponibili i dati in una cache in memoria ad alta velocità, migliorando le prestazioni e riducendo la latenza. Ciò riduce la necessità di accedere direttamente alle fonti di dati. La memorizzazione nella cache è disponibile sia per i resolver di unità che per quelli di pipeline.

AWS AppSync consente inoltre di comprimere le risposte delle API in modo che i contenuti del payload vengano caricati e scaricati più velocemente. Ciò riduce potenzialmente il carico sulle applicazioni e allo stesso tempo riduce potenzialmente i costi di trasferimento dei dati. Il comportamento di compressione è configurabile e può essere impostato a propria discrezione.

Consulta questa sezione per informazioni sulla definizione del comportamento desiderato della memorizzazione nella cache e della compressione lato server nell'API. AWS AppSync

Tipi di istanza

AWS AppSync ospita istanze Amazon ElastiCache (Redis OSS) nello stesso AWS account e AWS nella stessa regione dell'API. AWS AppSync

Sono disponibili i seguenti tipi di istanze ElastiCache (Redis OSS):

small

1 vCPU, 1,5 GiB RAM, prestazioni di rete da basse a moderate

medium

2 vCPU, 3 GiB RAM, prestazioni di rete da basse a moderate

large

2 vCPU, 12,3 GiB RAM, prestazioni di rete fino a 10 Gigabit

xlarge

4 vCPU, 25,05 GiB RAM, prestazioni di rete fino a 10 Gigabit

2xlarge

8 vCPU, 50,47 GiB RAM, prestazioni di rete fino a 10 Gigabit

4xlarge

16 vCPU, 101,38 GiB RAM, prestazioni di rete fino a 10 Gigabit

8xlarge

32 vCPU, 203,26 GiB RAM, prestazioni di rete a 10 Gigabit (non disponibile in tutte le regioni)

12xlarge

48 vCPU, 317,77 GiB RAM, prestazioni di rete a 10 Gigabit

Note

Storicamente, si specificava un tipo di istanza specifico (ad esempio). `t2.medium` A luglio 2020, questi tipi di istanze legacy continuano a essere disponibili, ma il loro utilizzo è obsoleto e sconsigliato. Ti consigliamo di utilizzare i tipi di istanza generici descritti qui.

Comportamento della cache

Di seguito sono riportati i comportamenti relativi alla memorizzazione nella cache:

Nessuno

Nessuna memorizzazione nella cache sul lato server.

Memorizzazione nella cache della richiesta completa

La memorizzazione completa delle richieste nella cache è un meccanismo che memorizza nella cache i risultati di esecuzione del resolver individualmente. Con questa impostazione, AWS AppSync memorizza nella cache l'esecuzione di tutti i resolver richiamati durante una richiesta, con ogni resolver memorizzato nella cache separatamente. I dati di ogni resolver vengono recuperati dalla relativa fonte di dati e popolano la cache fino alla scadenza del time to live (TTL). Per le successive richieste API, i risultati per ogni resolver specifico vengono restituiti dalla cache. Ciò significa che le fonti di dati non vengono contattate direttamente a meno che il TTL non sia scaduto. AWS AppSync utilizza il contenuto di `context.arguments` e `context.identity` maps come chiavi di memorizzazione nella cache per ogni resolver.

Memorizzazione nella cache dei resolver

Con questa impostazione, è necessario attivare esplicitamente ogni resolver per memorizzare nella cache le risposte. È possibile specificare un TTL e delle chiavi di memorizzazione nella

cache sul resolver. Le chiavi di memorizzazione nella cache che è possibile specificare sono le mappe di primo livello e `context.identity` i `context.arguments` campi `context.source` stringa di queste mappe. and/or Il valore TTL è obbligatorio, ma le chiavi di memorizzazione nella cache sono facoltative. Se non si specifica alcuna chiave di memorizzazione nella cache, le impostazioni predefinite sono i contenuti delle `context.arguments` mappe, e. `context.source` `context.identity`

Ad esempio, è possibile utilizzare le seguenti combinazioni:

- `context.arguments` e `context.source`
- `context.arguments` e `context.identity.sub`
- `context.arguments.id` o `context.arguments.InputType.id`
- `context.source.id` e `context.identity.sub`
- `context.identity.claims.username`

Quando si specifica solo un TTL e nessuna chiave di memorizzazione nella cache, il comportamento del resolver è lo stesso della memorizzazione nella cache completa delle richieste.

Memorizzazione nella cache a livello operativo

La memorizzazione nella cache a livello operativo archivia tutte le risposte alle operazioni di interrogazione GraphQL nel loro insieme. Se abilitata, le risposte alle query riuscite vengono memorizzate nella cache fino alla scadenza del relativo TTL, con una dimensione massima di risposta inseribile nella cache di 15 MB. Per le richieste di query successive con la stessa chiave di cache, le risposte verranno fornite direttamente dalla cache senza eseguire alcun resolver finché il TTL non è scaduto.

La chiave di cache per la memorizzazione nella cache a livello operativo viene generata utilizzando una combinazione dei seguenti elementi:

- Alcuni attributi del payload JSON della richiesta:
 - La stringa `query`
 - La `operationName` corda
 - La `variables` mappa
- La `context.identity` mappa (escluse `context.identity.sourceIp` le richieste IAM e Amazon Cognito)
- La `context.request.headers` mappa (escluse alcune intestazioni riservate elencate nella sezione successiva)

Il tipo di autorizzazione utilizzato dalla richiesta influirà anche sulla chiave della cache. Per le richieste autorizzate da IAM, la chiave cache includerà inoltre l'elenco delle risorse consentite e negate. Per le richieste autorizzate da Lambda, la chiave cache includerà inoltre l'elenco dei campi negati.

La chiave cache prenderà in considerazione tutte le intestazioni di richiesta presenti in `context.request.headers`, ad eccezione delle seguenti intestazioni riservate, che in genere sono uniche per richieste specifiche:

- `authorization`
- `cloudfront-forwarded-proto`
- `cloudfront-is-desktop-viewer`
- `cloudfront-is-mobile-viewer`
- `cloudfront-is-smarttv-viewer`
- `cloudfront-is-tablet-viewer`
- `cloudfront-viewer-asn`
- `cloudfront-viewer-country`
- `content-length`
- `host`
- `priority`
- `sec-ch-ua`
- `sec-ch-ua-mobile`
- `sec-ch-ua-platform`
- `via`
- `x-amz-cf-id`
- `x-amz-date`
- `x-amz-security-token`
- `x-amzn-appsync-is-vpce-request`
- `x-amzn-remote-ip`
- `x-amzn-requestid`
- `x-amzn-trace-id`

- `x-forwarded-for`

Tempo di utilizzo della cache

Questa impostazione definisce la quantità di tempo per archiviare le voci memorizzate nella cache. Il TTL massimo è di 3.600 secondi (1 ora), dopodiché le voci vengono eliminate automaticamente.

Crittografia cache

Quando si utilizza AWS AppSync la funzionalità di memorizzazione nella cache dei dati lato server, la crittografia a riposo e in transito è sempre abilitata per le nuove cache e non può essere disabilitata.

Per abilitare la crittografia su una cache API esistente, elimina la cache e ricreala.

Per invalidare le voci della cache, puoi effettuare una chiamata API flush cache utilizzando la AWS AppSync console o il comando `flush-cache` di AWS Command Line Interface AWS CLI.

Per ulteriori informazioni, consulta il tipo di [ApiCache](#) dati nell'API Reference. AWS AppSync

Eliminazione della cache

Quando configuri la memorizzazione AWS AppSync nella cache lato server, puoi configurare un TTL massimo. Questo valore definisce la quantità di tempo in cui le voci memorizzate nella cache vengono archiviate in memoria. In situazioni in cui è necessario rimuovere voci specifiche dalla cache, AWS AppSync è possibile utilizzare l'utilità `evictFromApiCache` extensions nella richiesta o nella risposta del resolver. (Ad esempio, quando i dati nelle fonti di dati sono cambiati e la voce della cache è ora obsoleta.) Per rimuovere un elemento dalla cache, è necessario conoscerne la chiave. Per questo motivo, se devi rimuovere gli elementi in modo dinamico, ti consigliamo di utilizzare la memorizzazione nella cache per resolver e di definire esplicitamente una chiave da utilizzare per aggiungere voci alla cache.

Eliminare una voce dalla cache

Per eliminare un elemento dalla cache, utilizzate l'utilità `evictFromApiCache` extensions. Specificate il nome del tipo e il nome del campo, quindi fornite un oggetto di elementi chiave-valore per creare la chiave della voce che desiderate eliminare. Nell'oggetto, ogni chiave rappresenta una voce valida dell'`context` oggetto utilizzato nell'elenco del resolver memorizzato nella cache. `cachingKey` Ogni valore è il valore effettivo utilizzato per costruire il valore della chiave. È

necessario inserire gli elementi nell'oggetto nello stesso ordine delle chiavi di memorizzazione nella cache nell'elenco del resolver memorizzato nella cache. `cachingKey`

Ad esempio, vedete lo schema seguente:

```
type Note {  
  id: ID!  
  title: String  
  content: String!  
}  
  
type Query {  
  getNote(id: ID!): Note  
}  
  
type Mutation {  
  updateNote(id: ID!, content: String!): Note  
}
```

In questo esempio, è possibile abilitare la memorizzazione nella cache per resolver, quindi abilitarla per la query. `getNote` Quindi, è possibile configurare la chiave di memorizzazione nella cache in modo che sia composta da. `[context.arguments.id]`

Quando si tenta di ottenere una `Note`, per creare la chiave della cache, AWS AppSync esegue una ricerca nella sua cache lato server utilizzando l'argomento `id` della query. `getNote`

Quando aggiorni una `Note`, devi eliminare la voce relativa alla nota specifica per assicurarti che la richiesta successiva la recuperi dalla fonte dati di backend. A tale scopo, è necessario creare un gestore di richieste.

L'esempio seguente mostra un modo per gestire lo sfratto utilizzando questo metodo:

```
import { util, Context } from '@aws-appsync/utils';  
import { update } from '@aws-appsync/utils/dynamodb';  
  
export function request(ctx) {  
  extensions.evictFromApiCache('Query', 'getNote', { 'ctx.args.id': ctx.args.id });  
  return update({ key: { id: ctx.args.id }, update: { context: ctx.args.content } });  
}  
  
export const response = (ctx) => ctx.result;
```

In alternativa, puoi anche gestire lo sfratto nel gestore delle risposte.

Quando la `updateNote` mutazione viene elaborata, AWS AppSync tenta di eliminare la voce. Se una voce viene cancellata con successo, la risposta contiene un `apiCacheEntriesDeleted` valore nell'`extensions` oggetto che mostra quante voci sono state eliminate:

```
"extensions": {  "apiCacheEntriesDeleted": 1}
```

Eliminare una voce della cache in base all'identità

È possibile creare chiavi di memorizzazione nella cache basate su più valori dell'oggetto. `context`

Ad esempio, prendi lo schema seguente che utilizza i pool di utenti di Amazon Cognito come modalità di autenticazione predefinita ed è supportato da un'origine dati Amazon DynamoDB:

```
type Note {
  id: ID! # a slug; e.g.: "my-first-note-on-graphql"
  title: String
  content: String!
}

type Query {
  getNote(id: ID!): Note
}

type Mutation {
  updateNote(id: ID!, content: String!): Note
}
```

I tipi di Note oggetto vengono salvati in una tabella DynamoDB. La tabella ha una chiave composta che utilizza il nome utente di Amazon Cognito come chiave primaria e il `id` (uno slug) di Note come chiave di partizione. Si tratta di un sistema multi-tenant che consente a più utenti di ospitare e aggiornare i propri Note oggetti privati, che non vengono mai condivisi.

Poiché si tratta di un sistema ad alta intensità di lettura, la `getNote` query viene memorizzata nella cache utilizzando la memorizzazione nella cache per resolver, con la chiave di memorizzazione composta da. `[context.identity.username, context.arguments.id]` Quando a Note viene aggiornato, puoi rimuovere la voce relativa a quello specifico. Note È necessario aggiungere i componenti dell'oggetto nello stesso ordine in cui sono specificati nell'elenco del resolver.

`cachingKeys`

L'esempio seguente lo dimostra:

```
import { util, Context } from '@aws-appsync/utils';
import { update } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  extensions.evictFromApiCache('Query', 'getNote', {
    'ctx.identity.username': ctx.identity.username,
    'ctx.args.id': ctx.args.id,
  });
  return update({ key: { id: ctx.args.id }, update: { context: ctx.args.content } });
}

export const response = (ctx) => ctx.result;
```

Un sistema di backend può anche aggiornare Note ed eliminare la voce. Ad esempio, prendiamo questa mutazione:

```
type Mutation {
  updateNoteFromBackend(id: ID!, content: String!, username: ID!): Note @aws_iam
}
```

È possibile eliminare la voce, ma aggiungere i componenti della chiave di memorizzazione nella cache all'oggetto. `cachingKeys`

Nell'esempio seguente, lo sfratto avviene nella risposta del resolver:

```
import { util, Context } from '@aws-appsync/utils';
import { update } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  return update({ key: { id: ctx.args.id }, update: { context: ctx.args.content } });
}

export function response(ctx) {
  extensions.evictFromApiCache('Query', 'getNote', {
    'ctx.identity.username': ctx.args.username,
    'ctx.args.id': ctx.args.id,
  });
  return ctx.result;
}
```

Nei casi in cui i dati del backend sono stati aggiornati all'esterno AWS AppSync, è possibile rimuovere un elemento dalla cache richiamando una mutazione che utilizza una fonte di dati. NONE

Compressione delle risposte API

AWS AppSync consente ai clienti di richiedere payload compressi. Se richiedi, le risposte API vengono compresse e restituite in risposta alle richieste che indicano che il contenuto compresso è preferito. Le risposte API compresse si caricano più velocemente, i contenuti vengono scaricati più velocemente e anche i costi di trasferimento dei dati possono essere ridotti.

Note

La compressione è disponibile per tutte le novità APIs create dopo il 1° giugno 2020. AWS AppSync [comprime](#) gli oggetti nel miglior modo possibile. In rari casi, AWS AppSync può saltare la compressione in base a una serie di fattori, inclusa la capacità attuale.

AWS AppSync può comprimere dimensioni del payload delle query GraphQL comprese tra 1.000 e 10.000.000 di byte. Per abilitare la compressione, un client deve inviare l'intestazione con il valore. Accept-Encoding gzip La compressione può essere verificata controllando il valore dell'Content-Encoding intestazione nella risposta (gzip).

Per impostazione predefinita, il query explorer nella AWS AppSync console imposta automaticamente il valore dell'intestazione nella richiesta. Se esegui una query con una risposta sufficientemente ampia, la compressione può essere confermata utilizzando gli strumenti di sviluppo del browser.

Configurazione di nomi di dominio personalizzati per GraphQL e in tempo reale APIs

Con AWS AppSync, puoi utilizzare nomi di dominio personalizzati per configurare un unico dominio memorabile che funzioni sia per il tuo GraphQL che in tempo reale. APIs

In altre parole, puoi utilizzare un endpoint semplice e facile da ricordare URLs con nomi di dominio a tua scelta creando nomi di dominio personalizzati da associare al tuo account. AWS AppSync APIs

Quando configuri un' AWS AppSync API, vengono forniti due endpoint:

AWS AppSync Endpoint GraphQL:

`https://example1234567890000.appsync-api.us-east-1.amazonaws.com/graphql`

AWS AppSync endpoint in tempo reale:

```
wss://example12345678900000.appsync-realtime-api.us-east-1.amazonaws.com/graphql
```

Con i nomi di dominio personalizzati, puoi interagire con entrambi gli endpoint utilizzando un unico dominio. Ad esempio, se configuri `api.example.com` come dominio personalizzato, puoi interagire sia con gli endpoint GraphQL che con quelli in tempo reale utilizzando questi: URLs

AWS AppSync endpoint GraphQL di dominio personalizzato:

```
https://api.example.com/graphql
```

AWS AppSync endpoint in tempo reale con dominio personalizzato:

```
wss://api.example.com/graphql/realtime
```

Note

AWS AppSync APIs supporta solo TLS 1.2 e TLS 1.3 per nomi di dominio personalizzati.

Registrazione e configurazione di un nome di dominio

Per configurare nomi di dominio personalizzati per il tuo AWS AppSync APIs, devi disporre di un nome di dominio Internet registrato. Puoi registrare un dominio Internet utilizzando Amazon Route 53 domain registration o un registrar di domini di terze parti a tua scelta. Per ulteriori informazioni su Route 53, consulta [What is Amazon Route 53?](#) nella Amazon Route 53 Developer Guide.

Il nome di dominio personalizzato di un'API può essere il nome di un sottodominio o il dominio principale (noto anche come «apice di zona») di un dominio Internet registrato. Dopo aver creato un nome di dominio personalizzato in AWS AppSync, devi creare o aggiornare il record di risorse del tuo provider DNS da mappare all'endpoint API. Senza questa mappatura, le richieste API associate al nome di dominio personalizzato non possono arrivare. AWS AppSync

Creazione di un nome di dominio personalizzato in AWS AppSync

La creazione di un nome di dominio personalizzato per un' AWS AppSync API configura una Amazon CloudFront distribuzione. È necessario configurare un record DNS per mappare il nome di dominio personalizzato al nome di dominio di CloudFront distribuzione. Questa mappatura

è necessaria per indirizzare le richieste API associate al nome di dominio personalizzato AWS AppSync attraverso la distribuzione CloudFront mappata. Devi inoltre fornire un certificato per il nome di dominio personalizzato.

Per configurare il nome di dominio personalizzato o aggiornarne il certificato, devi disporre dell'autorizzazione per aggiornare CloudFront le distribuzioni e descrivere il certificato AWS Certificate Manager (ACM) che intendi utilizzare. Per concedere queste autorizzazioni, allega la seguente dichiarazione politica AWS Identity and Access Management (IAM) a un utente, gruppo o ruolo IAM nel tuo account:

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowUpdateDistributionForAppSyncCustomDomainName",
      "Effect": "Allow",
      "Action": [
        "cloudfront:updateDistribution"
      ],
      "Resource": [
        "*"
      ]
    },
    {
      "Sid": "AllowDescribeCertificateForAppSyncCustomDomainName",
      "Effect": "Allow",
      "Action": "acm:DescribeCertificate",
      "Resource": "arn:aws:acm:us-east-1:111122223333:certificate/
certificate-id"
    }
  ]
}
```

AWS AppSync supporta nomi di dominio personalizzati sfruttando Server Name Indication (SNI) sulla distribuzione CloudFront. Per ulteriori informazioni sull'utilizzo di nomi di dominio personalizzati su una CloudFront distribuzione, incluso il formato di certificato richiesto e la lunghezza massima della chiave del certificato, consulta [Using HTTPS with CloudFront](#) nella Amazon CloudFront Developer Guide.

Per configurare un nome di dominio personalizzato come nome host dell'API, il proprietario dell'API deve fornire un certificato ACM pubblico o importato nella us-east-1 AWS regione (Stati Uniti orientali (Virginia settentrionale)) che copre il nome di dominio personalizzato. [Per ulteriori informazioni su ACM, consulta What is? AWS Certificate Manager](#) nella Guida per l'AWS Certificate Manager utente.

Nomi di dominio personalizzati Wildcard in AWS AppSync

AWS AppSync supporta nomi di dominio personalizzati con caratteri wildcard. Per configurare un nome di dominio personalizzato con caratteri jolly, specifica un carattere jolly (*) come primo sottodominio di un dominio personalizzato. Rappresenta tutti i possibili sottodomini del dominio principale. Ad esempio, il nome di dominio personalizzato con caratteri jolly *.example.com restituisce sottodomini quali a.example.com, b.example.com e c.example.com. Tutti questi sottodomini vengono indirizzati allo stesso dominio.

Per utilizzare un nome di dominio personalizzato con caratteri jolly in AWS AppSync, devi fornire un certificato emesso da ACM contenente un nome jolly in grado di proteggere diversi siti nello stesso dominio. Per ulteriori informazioni, consulta le [caratteristiche del certificato ACM](#) nella Guida per l'utente AWS Certificate Manager.

Operazioni di controllo delle versioni, rilevamento dei conflitti e sincronizzazione per le sorgenti dati DynamoDB in AWS AppSync

AWS AppSync le funzionalità avanzate di gestione dei dati per DynamoDB sfruttano tre funzionalità chiave: origini dati con versione, rilevamento e risoluzione dei conflitti e operazioni di sincronizzazione. Questi strumenti consentono applicazioni robuste e scalabili che gestiscono in modo efficiente le modifiche e la sincronizzazione simultanee dei dati in ambienti distribuiti.

Le fonti di dati con versioni diverse costituiscono la base del sistema di gestione dei dati di AWS AppSync The Data Management. Questa funzionalità migliora automaticamente gli elementi di DynamoDB con metadati di controllo delle versioni, registra le modifiche apportate AWS AppSync dalle mutazioni a una tabella Delta e mantiene le «lapidi» per gli elementi eliminati. Gli sviluppatori possono configurare i periodi di conservazione degli elementi eliminati e dei log delle modifiche, ottimizzando l'archiviazione e garantendo al contempo la coerenza dei dati. Le fonti di dati con versioni semplificano l'implementazione delle operazioni di rilevamento e sincronizzazione dei conflitti, fornendo una solida base per la gestione avanzata dei dati.

I meccanismi di rilevamento e risoluzione dei conflitti salvaguardano l'integrità dei dati quando si verificano scritture simultanee. AWS AppSync offre tre strategie: Optimistic Concurrency, Automerge e Risoluzione basata su Lambda. Optimistic Concurrency rifiuta le mutazioni contrastanti, consentendo ai clienti di riprovare con dati aggiornati. Automerge risolve automaticamente i conflitti in base ai tipi di dati, alla fusione di elenchi, all'esecuzione di unioni di set e alla conservazione dei valori scalari esistenti. La risoluzione basata su Lambda consente una logica personalizzata per scenari di conflitto complessi. Queste opzioni offrono agli sviluppatori flessibilità nella gestione dei conflitti di dati, garantendo la coerenza tra i sistemi distribuiti.

Le operazioni di sincronizzazione consentono il recupero e l'aggiornamento efficienti dei dati nelle applicazioni client. Questa funzionalità consente ai client di recuperare tutti i risultati da una tabella DynamoDB e successivamente recuperare solo i dati modificati dall'ultima query. AWS AppSync determina se accedere alla tabella Base o alla tabella Delta in base al token di sincronizzazione fornito, ottimizzando le prestazioni e riducendo il trasferimento dei dati.

Argomenti

- [Controllo delle versioni delle sorgenti dati DynamoDB in AWS AppSync](#)
- [Rilevamento e risoluzione dei conflitti in AWS AppSync](#)
- [Utilizzo delle operazioni di sincronizzazione di DynamoDB su sorgenti dati con versione in AWS AppSync](#)

Controllo delle versioni delle sorgenti dati DynamoDB in AWS AppSync

AWS AppSync attualmente supporta il controllo delle versioni su sorgenti dati DynamoDB. Le operazioni di rilevamento dei conflitti, risoluzione dei conflitti e sincronizzazione richiedono un'origine dati Versioned. Quando abiliti il controllo delle versioni su un'origine dati, automaticamente: AWS AppSync

- Migliorare gli elementi con i metadati del controllo delle versioni degli oggetti.
- Registra le modifiche apportate agli elementi con AWS AppSync mutazioni in una tabella Delta.
- Gestire gli elementi eliminati nella tabella Base con una "rimozione definitiva" per un periodo di tempo configurabile.

Configurazione dell'origine dati con versione

Quando si attivano le versioni multiple in un'origine dati DynamoDB, specificare i seguenti campi:

BaseTableTTL

Il numero di minuti per mantenere gli elementi eliminati nella tabella Base con una "rimozione definitiva", un campo di metadati che indica che l'elemento è stato eliminato. È possibile impostare questo valore su 0 se si desidera che gli elementi vengano rimossi immediatamente quando vengono eliminati. Questo campo è obbligatorio.

DeltaSyncTableName

Il nome della tabella in cui vengono archiviate le modifiche apportate agli elementi con AWS AppSync mutazioni. Questo campo è obbligatorio.

DeltaSyncTableTTL

Numero di minuti per conservare gli elementi nella tabella Delta. Questo campo è obbligatorio.

Registrazione della tabella Delta Sync

AWS AppSync attualmente supporta Delta Sync Logging for mutations using `PutItemUpdateItem`, e le operazioni `DynamoDBDeleteItem`.

Quando una AWS AppSync mutazione modifica un elemento in un'origine dati con versione, un record di tale modifica verrà archiviato in una tabella Delta ottimizzata per gli aggiornamenti incrementali. Puoi scegliere di utilizzare diverse tabelle Delta (ad esempio una per tipo, una per area di dominio) per altre fonti di dati con versione o una singola tabella Delta per la tua API. AWS AppSync sconsiglia di utilizzare una singola tabella Delta per più tabelle per APIs evitare la collisione delle chiavi primarie.

Lo schema richiesto per questa tabella è il seguente:

ds_pk

Valore stringa utilizzato come chiave di partizione. È costruito concatenando il nome della fonte di dati di base e il formato ISO 8601 della data in cui è avvenuta la modifica (ad es.).
`Comments:2019-01-01`

Quando il *customPartitionKey* flag del modello di mappatura VTL è impostato come nome di colonna della chiave di partizione (vedi [Resolver Mapping Template Reference for DynamoDB](#) nella Developer Guide), AWS AppSync il formato *ds_pk* delle modifiche e la stringa viene costruita aggiungendole il valore della chiave di partizione nel nuovo record nella tabella Base.

Ad esempio, se il record nella tabella Base ha un valore di chiave di partizione 1a e un valore di chiave di ordinamento di 2b, il nuovo valore della stringa sarà: `Comments:2019-01-01:1a`.

ds_sk

Un valore stringa utilizzato come chiave di ordinamento. Viene costruito concatenando il formato ISO 8601 dell'ora in cui è avvenuta la modifica, la chiave primaria dell'elemento e la versione dell'elemento. La combinazione di questi campi garantisce l'unicità di ogni voce nella tabella Delta (ad esempio, per l'ora di, l'ID 09:30:00 e la versione di 1a2, questo sarebbe). `09:30:00:1a:2`

Quando il *customPartitionKey* flag del modello di mappatura VTL è impostato sul nome della colonna della chiave di partizione (vedi [Resolver Mapping Template Reference for DynamoDB](#) nella Developer Guide), AWS AppSync il formato *ds_sk* delle modifiche e la stringa viene costruita sostituendo il valore della chiave combinata con il valore della chiave di ordinamento nella tabella Base. Utilizzando l'esempio precedente, se il record nella tabella Base ha un valore di chiave di partizione 1a e un valore di chiave di ordinamento di 2b, il nuovo valore della stringa sarà: `09:30:00:2b:3`.

_ttl

Un valore numerico che memorizza il timestamp, in secondi dall'epoca, quando un elemento deve essere rimosso dalla tabella Delta. Questo valore viene determinato aggiungendo il valore `DeltaSyncTableTTL` configurato nell'origine dati nel momento in cui si è verificata la modifica. Questo campo deve essere configurato come Attributo TTL DynamoDB.

Il ruolo IAM configurato per l'utilizzo con la tabella Base deve contenere anche le autorizzazioni per operare nella tabella Delta. In questo esempio, viene visualizzato il criterio delle autorizzazioni per una tabella Base denominata `Comments` e una tabella Delta denominata `ChangeLog`:

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:DeleteItem",
        "dynamodb:GetItem",
        "dynamodb:PutItem",
```

```

        "dynamodb:Query",
        "dynamodb:Scan",
        "dynamodb:UpdateItem"
    ],
    "Resource": [
        "arn:aws:dynamodb:us-east-1:000000000000:table/Comments",
        "arn:aws:dynamodb:us-east-1:000000000000:table/Comments/*",
        "arn:aws:dynamodb:us-east-1:000000000000:table/ChangeLog",
        "arn:aws:dynamodb:us-east-1:000000000000:table/ChangeLog/*"
    ]
}
]
}
```

Metadati dell'origine dati con versione

AWS AppSync gestisce i campi di metadati sulle fonti di **Versioned** dati per tuo conto. La modifica di questi campi può causare errori nell'applicazione o perdita di dati. Questi campi includono:

`_version`

Un contatore monotonicamente crescente che viene aggiornato ogni volta che si apporta una modifica a un elemento.

`_lastChangedAt`

Un valore numerico che memorizza il timestamp, in millisecondi dall'epoca, al momento dell'ultima modifica di un elemento.

`_deleted`

Un valore booleano "contrassegnato per la rimozione definitiva" che indica che un elemento è stato eliminato. Questo può essere utilizzato dalle applicazioni per rimuovere gli elementi eliminati dagli archivi dati locali.

`_ttl`

Un valore numerico che memorizza il timestamp, in secondi dall'epoca, quando un elemento deve essere rimosso dall'origine dati sottostante.

`ds_pk`

Valore stringa utilizzato come chiave di partizione per le tabelle Delta.

ds_sk

Valore stringa utilizzato come chiave di ordinamento per le tabelle Delta.

gsi_ds_pk

Un attributo di valore di stringa generato per supportare un indice secondario globale come chiave di partizione. Verrà incluso solo se entrambi i *populateIndexFields* flag *customPartitionKey* e sono abilitati nel modello di mappatura VTL (vedi [Resolver Mapping Template Reference for DynamoDB nella Developer Guide](#)).AWS AppSync Se abilitato, il valore verrà costruito concatenando il nome della fonte di dati di Base e il formato ISO 8601 della data in cui è avvenuta la modifica (ad esempio se la tabella Base è denominata Comments, questo record verrà impostato come). `Comments:2019-01-01`

gsi_ds_sk

Un attributo di valore di stringa generato per supportare un indice secondario globale come chiave di ordinamento. Verrà incluso solo se entrambi i *populateIndexFields* flag *customPartitionKey* e sono abilitati nel modello di mappatura VTL (vedi [Resolver Mapping Template Reference for DynamoDB nella Developer Guide](#)).AWS AppSync Se abilitato, il valore verrà costruito concatenando il formato ISO 8601 dell'ora in cui è avvenuta la modifica, la chiave di partizione dell'elemento nella tabella Base, la chiave di ordinamento dell'elemento nella tabella Base e la versione dell'elemento (ad esempio per un periodo di `09:30:00`, un valore della chiave di partizione di `1a`, un valore della chiave di ordinamento di `2b` e la versione di `3`, questo sarebbe `09:30:00:1a#2b:3`).

Questi campi di metadati influiranno sulla dimensione complessiva degli elementi nell'origine dati sottostante. AWS AppSync consiglia di riservare una dimensione massima di archiviazione di 500 byte o più della chiave primaria per i metadati delle origini dati con versioni diverse durante la progettazione dell'applicazione. Per utilizzare questi metadati nelle applicazioni client, includere i campi `_version`, `_lastChangedAt` e `_deleted` nei tipi GraphQL e nel set di selezione per le mutazioni.

Rilevamento e risoluzione dei conflitti in AWS AppSync

Quando vengono eseguite scritture simultanee con AWS AppSync, è possibile configurare le strategie di rilevamento e risoluzione dei conflitti per gestire gli aggiornamenti in modo appropriato. Il rilevamento dei conflitti determina se la mutazione è in conflitto con l'elemento scritto effettivo nell'origine dati. Il rilevamento dei conflitti viene abilitato impostando il `SyncConfig` valore nel `conflictDetection` campo su `VERSION`

La risoluzione dei conflitti è l'azione che viene eseguita nel caso in cui venga rilevato un conflitto. Ciò viene determinato impostando il campo Conflict Handler nel SyncConfig. Esistono tre strategie di risoluzione dei conflitti:

- OPTIMISTIC_CONCURRENCY
- AUTOMERGE
- LAMBDA

Le versioni vengono incrementate automaticamente AWS AppSync durante le operazioni di scrittura e non devono essere modificate dai client o dall'esterno di un resolver configurato con un'origine dati abilitata alla versione. Ciò cambierà il comportamento di coerenza del sistema e potrebbe comportare la perdita di dati.

Concorrenza ottimistica

Optimistic Concurrency è una strategia di risoluzione dei conflitti che AWS AppSync prevede fonti di dati con versioni diverse. Quando il risolutore dei conflitti è impostato su Optimistic Concurrency, se viene rilevata una mutazione in ingresso per avere una versione diversa dalla versione effettiva dell'oggetto, il gestore dei conflitti rifiuterà semplicemente la richiesta in ingresso. All'interno della risposta GraphQL, verrà fornito l'elemento esistente sul server con la versione più recente. Il client dovrebbe quindi gestire questo conflitto localmente e riprovare la mutazione con la versione aggiornata dell'elemento.

Si fonde automaticamente

Automerge offre agli sviluppatori un modo semplice per configurare una strategia di risoluzione dei conflitti senza scrivere logica lato client per unire manualmente i conflitti che non erano in grado di essere gestiti da altre strategie. Automerge aderisce a un set di regole rigoroso quando si uniscono i dati per risolvere i conflitti. I principi di Automerge ruotano attorno al tipo di dati sottostante del campo GraphQL. Essi sono i seguenti:

- Conflitto su un campo scalare: scalare GraphQL o qualsiasi campo che non sia una raccolta (ad esempio List, Set, Map). Rifiuto del valore in ingresso per il campo scalare e selezione del valore esistente nel server.
- Conflitto su un elenco: il tipo GraphQL e il tipo di database sono elenchi. Concatenamento dell'elenco in entrata con l'elenco esistente nel server. I valori di elenco nella mutazione in arrivo verranno aggiunti alla fine dell'elenco nel server. I valori duplicati verranno mantenuti.

- Conflitto su un set: il tipo GraphQL è un elenco e il tipo di database è un set. Applicazione di un insieme unione utilizzando il set in ingresso e il set esistente nel server. Questo aderisce alle proprietà di un set, vale a dire nessuna voce duplicata.
- Quando una mutazione in arrivo aggiunge un nuovo campo all'elemento o viene creata su un campo con il valore `null`, uniscilo all'elemento esistente.
- Conflitto su una mappa: quando il tipo di dati sottostante nel database è una mappa (ad esempio documento chiave-valore), applicare le regole di cui sopra mentre analizza ed elabora ciascuna proprietà della mappa.

Automerge è progettato per rilevare, unire e riprovare automaticamente le richieste con una versione aggiornata, assolvendo il client dalla necessità di unire manualmente i dati in conflitto.

Per mostrare un esempio di come Automerge gestisce un conflitto su un tipo scalare. Useremo il seguente record come punto di partenza.

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 5,
  "_version" : 4
}
```

Ora una mutazione in arrivo potrebbe tentare di aggiornare l'elemento ma con una versione precedente poiché il client non è ancora sincronizzato con il server. Dovrebbe essere simile a questo:

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 55,
  "_version" : 2
}
```

Si noti la versione obsoleta di 2 nella richiesta in entrata. Durante questo flusso, Automerge unirà i dati rifiutando l'aggiornamento del campo "jersey" a "55" e manterrà il valore a "5" con conseguente salvataggio della seguente immagine dell'elemento nel server.

```
{
  "id" : 1,
```

```
"name" : "Nadia",
"jersey" : 5,
"_version" : 5 # version is incremented every time automerge performs a merge that is
stored on the server.
}
```

Dato lo stato dell'elemento mostrato sopra alla versione 5, ora supponiamo che una mutazione in entrata tenti di cambiare l'elemento con la seguente immagine:

```
{
  "id" : 1,
  "name" : "Shaggy",
  "jersey" : 5,
  "interests" : ["breakfast", "lunch", "dinner"] # underlying data type is a Set
  "points": [24, 30, 27] # underlying data type is a List
  "_version" : 3
}
```

La mutazione in entrata ha tre punti di interesse. Il nome, uno scalare, è stato modificato ma sono stati aggiunti due nuovi campi "interessi", un Set, "punti" e un Elenco. In questo scenario, verrà rilevato un conflitto a causa della mancata corrispondenza della versione. Automerge aderisce alle sue proprietà e rifiuta la modifica del nome data la sua natura scalare e l'add-on ai campi non in conflitto. In questo modo, l'elemento che viene salvato nel server viene visualizzato come segue.

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 5,
  "interests" : ["breakfast", "lunch", "dinner"] # underlying data type is a Set
  "points": [24, 30, 27] # underlying data type is a List
  "_version" : 6
}
```

Con l'immagine aggiornata dell'elemento con la versione 6, ora supponiamo che una mutazione in entrata (con un'altra mancata corrispondenza della versione) cerchi di trasformare l'elemento nel seguente modo:

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 5,
```



```

"interests" : ["breakfast", "lunch", "brunch"] # underlying data type is a Set
"points": [30, 35] # underlying data type is a List
"_version" : 5
}

```

Qui osserviamo che il campo in entrata per "interessi" ha un valore duplicato che esiste nel server e due nuovi valori. In questo caso, poiché il tipo di dati sottostante è un Set, Automerge combinerà i valori esistenti nel server con quelli nella richiesta in arrivo e rimuoverà eventuali duplicati. Allo stesso modo c'è un conflitto nel campo "punti" in cui c'è un valore duplicato e un nuovo valore. Ma poiché il tipo di dati sottostante qui è un elenco, Automerge aggiungerà semplicemente tutti i valori nella richiesta in entrata alla fine dei valori già esistenti nel server. L'immagine risultante unita memorizzata sul server apparirà come segue:

```

{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 5,
  "interests" : ["breakfast", "lunch", "dinner", "brunch"] # underlying data type is a Set
  "points": [24, 30, 27, 30, 35] # underlying data type is a List
  "_version" : 7
}

```

Ora supponiamo che l'elemento memorizzato nel server appaia come segue, nella versione 8.

```

{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 5,
  "interests" : ["breakfast", "lunch", "dinner", "brunch"] # underlying data type is a Set
  "points": [24, 30, 27, 30, 35] # underlying data type is a List
  "stats": {
    "ppg": "35.4",
    "apg": "6.3"
  }
  "_version" : 8
}

```

Ma supponiamo anche che una richiesta in entrata tenti di aggiornare l'elemento con la seguente immagine, ancora una volta con una mancata corrispondenza della versione:

```
{
  "id" : 1,
  "name" : "Nadia",
  "stats": {
    "ppg": "25.7",
    "rpg": "6.9"
  }
  "_version" : 3
}
```

Ora, in questo scenario, possiamo vedere che i campi che già esistono nel server sono mancanti (interessi, punti, jersey). Inoltre, il valore per "ppg" all'interno della mappa "statistiche" viene modificato, viene aggiunto un nuovo valore "rpg" e "apg" viene omissso. Automerge conserva i campi che sono stati omissi (nota: se i campi sono destinati a essere rimossi, allora la richiesta deve essere riprovata con la versione corrispondente), e quindi non andranno persi. Applicherà anche le stesse regole ai campi all'interno delle mappe e quindi la modifica a "ppg" verrà rifiutata mentre "apg" è conservato e "rpg", un nuovo campo, viene aggiunto. L'elemento risultante memorizzato nel server verrà ora visualizzato come:

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 5,
  "interests" : ["breakfast", "lunch", "dinner", "brunch"] # underlying data type is a Set
  "points": [24, 30, 27, 30, 35] # underlying data type is a List
  "stats": {
    "ppg": "35.4",
    "apg": "6.3",
    "rpg": "6.9"
  }
  "_version" : 9
}
```

Lambda

Esistono diverse strategie di risoluzione Lambda tra cui scegliere:

- **RESOLVE**: sostituisce l'articolo esistente con un nuovo articolo fornito nel payload di risposta. È possibile riprovare la stessa operazione solo su un singolo elemento alla volta. Attualmente supportato per DynamoDB PutItem e UpdateItem.

- **REJECT**: rifiuta la mutazione e restituisce un errore con l'elemento esistente nella risposta GraphQL. Attualmente supportato per DynamoDB PutItem, UpdateItem e DeleteItem.
- **REMOVE**: rimuove l'elemento esistente. Attualmente supportato per DynamoDB DeleteItem.

La richiesta di invocazione Lambda

Il AWS AppSync resolver DynamoDB richiama la funzione Lambda specificata in.

LambdaConflictHandlerArn Utilizza lo stesso service-role-arn configurato per l'origine dati.

Il payload dell'invocazione ha la seguente struttura:

```
{
  "newItem": { ... },
  "existingItem": {... },
  "arguments": { ... },
  "resolver": { ... },
  "identity": { ... }
}
```

I campi sono definiti come segue:

newItem

L'elemento di anteprima, se la mutazione è riuscita.

existingItem

L'elemento attualmente risiede nella tabella DynamoDB.

arguments

Gli argomenti della mutazione GraphQL.

resolver

Informazioni sul resolver AWS AppSync .

identity

Informazioni sul chiamante. Questo campo è impostato su null, se l'accesso avviene con chiave API.

Esempio di payload:

```
{
  "newItem": {
    "id": "1",
    "author": "Jeff",
    "title": "Foo Bar",
    "rating": 5,
    "comments": ["hello world"],
  },
  "existingItem": {
    "id": "1",
    "author": "Foo",
    "rating": 5,
    "comments": ["old comment"]
  },
  "arguments": {
    "id": "1",
    "author": "Jeff",
    "title": "Foo Bar",
    "comments": ["hello world"]
  },
  "resolver": {
    "tableName": "post-table",
    "awsRegion": "us-west-2",
    "parentType": "Mutation",
    "field": "updatePost"
  },
  "identity": {
    "accountId": "123456789012",
    "sourceIp": "x.x.x.x",
    "username": "AIDAAAAAAAAAAAAAAAAAAAA",
    "userArn": "arn:aws:iam::123456789012:user/appsync"
  }
}
```

La risposta all'invocazione Lambda

Per PutItem e risoluzione UpdateItem dei conflitti

Rifiuto della mutazione (RESOLVE). La risposta deve essere nel seguente formato.

```
{
  "action": "RESOLVE",
  "item": { ... }
```

```
}
```

Il campo `item` rappresenta un oggetto che verrà utilizzato per sostituire l'elemento esistente nell'origine dati sottostante. La chiave primaria e i metadati di sincronizzazione verranno ignorati se inclusi in `item`.

Rifiuto della mutazione (REJECT). La risposta deve essere nel seguente formato.

```
{
  "action": "REJECT"
}
```

Per la risoluzione dei conflitti `DeleteItem`

REMOVE l'articolo. La risposta deve essere nel seguente formato.

```
{
  "action": "REMOVE"
}
```

Rifiuto della mutazione (REJECT). La risposta deve essere nel seguente formato.

```
{
  "action": "REJECT"
}
```

La funzione Lambda di esempio qui sotto controlla chi effettua la chiamata e il nome del resolver. Se è stato creato da `jeffTheAdmin`, REMOVE l'oggetto per il `DeletePost` resolver o il conflitto con un nuovo elemento per RESOLVE i resolver. Update/Put In caso contrario, la mutazione è REJECT.

```
exports.handler = async (event, context, callback) => {
  console.log("Event: " + JSON.stringify(event));

  // Business logic goes here.
  var response;
  if ( event.identity.user == "jeffTheAdmin" ) {
    let resolver = event.resolver.field;

    switch(resolver) {
      case "deletePost":
```

```
        response = {
            "action" : "REMOVE"
        }
        break;

    case "updatePost":
    case "createPost":
        response = {
            "action" : "RESOLVE",
            "item": event.newItem
        }
        break;
    default:
        response = { "action" : "REJECT" };
    }
} else {
    response = { "action" : "REJECT" };
}

console.log("Response: " + JSON.stringify(response));
return response;
}
```

Errori

Di seguito è riportato un elenco di possibili errori che possono verificarsi durante un processo di risoluzione dei conflitti:

ConflictUnhandled

Rilevamento dei conflitti individua una mancata corrispondenza della versione e il gestore dei conflitti rifiuta la mutazione.

Esempio: risoluzione dei conflitti con un gestore di conflitti di Optimistic Concurrency. Oppure, il gestore di conflitti Lambda è stato restituito con REJECT.

ConflictError

Si verifica un errore interno quando si tenta di risolvere un conflitto.

Esempio: il gestore di conflitti Lambda ha restituito una risposta non valida. In alternativa, non è possibile richiamare il gestore di conflitti Lambda perché la risorsa fornita `LambdaConflictHandlerArn` non viene trovata.

MaxConflicts

Sono stati raggiunti tentativi massimi per la risoluzione dei conflitti.

Esempio: troppe richieste simultanee sullo stesso oggetto. Prima che il conflitto venga risolto, l'oggetto viene aggiornato a una nuova versione da un altro client.

BadRequest

Il client tenta di aggiornare i campi dei metadati (`_version`, `_ttl`, `_lastChangedAt`, `_deleted`).

Esempio: il client tenta `_version` di aggiornare un oggetto con una mutazione di aggiornamento.

DeltaSyncWriteError

Impossibile scrivere il record di sincronizzazione delta.

Esempio: la mutazione è riuscita, ma si è verificato un errore interno durante il tentativo di scrivere nella tabella di sincronizzazione delta.

InternalFailure

Si è verificato un errore interno.

UnsupportedOperation

Operazione non supportata ". **X** Datasource Versioning supporta solo le seguenti operazioni (TransactGetItems,,, ScanPutItem, QueryBatchGetItem,,,, GetItem SyncDeleteItem). UpdateItem

Esempio: utilizzo di determinate operazioni di transazione e batch con conflitto abilitato. detection/resolution Queste operazioni non sono attualmente supportate.

CloudWatch Registri

Se un' AWS AppSync API ha abilitato CloudWatch i registri con le impostazioni di registrazione impostate su Registri a livello di campo `enabled` e a livello di registro per i registri a livello di campo impostate su `ALL`, AWS AppSync invierà le informazioni di rilevamento e risoluzione dei conflitti al gruppo di log. Per informazioni sul formato dei messaggi di registro, consulta la [documentazione relativa al rilevamento conflitti e alla registrazione di sincronizzazione](#).

Utilizzo delle operazioni di sincronizzazione di DynamoDB su sorgenti dati con versione in AWS AppSync

Le fonti di dati con versione supportano Sync operazioni che consentono di recuperare tutti i risultati da una tabella DynamoDB e quindi ricevere solo i dati modificati dall'ultima query (gli aggiornamenti delta). Quando AWS AppSync riceve una richiesta di Sync operazione, utilizza i campi specificati nella richiesta per determinare se è necessario accedere alla tabella Base o alla tabella Delta.

- Se il `lastSync` campo non è specificato, viene eseguita una Scan nella tabella Base.
- Se il campo `lastSync` è specificato, ma il valore è prima del `current moment` - `DeltaSyncTTL`, viene eseguita una Scan sulla tabella di base.
- Se il campo `lastSync` è specificato e il valore è attivo o dopo `current moment` - `DeltaSyncTTL`, viene eseguita una Query sulla tabella Delta.

AWS AppSync restituisce il `startedAt` campo al modello di mappatura delle risposte per tutte le Sync operazioni. Il campo `startedAt` è il momento, in millisecondi dall'epoca, in cui è iniziata l'operazione Sync che è possibile memorizzare localmente e utilizzare in un'altra richiesta. Se un token di paginazione è stato incluso nella richiesta, questo valore sarà lo stesso di quello restituito dalla richiesta per la prima pagina di risultati.

Per informazioni sul formato per i modelli di mappatura Sync, consulta il [riferimento del modello di mappatura](#).

Utilizzo CloudWatch per monitorare e registrare i dati dell'API GraphQL

Puoi registrare ed eseguire il debug dell'API GraphQL CloudWatch utilizzando metriche CloudWatch e log. Questi strumenti consentono agli sviluppatori di monitorare le prestazioni, risolvere i problemi e ottimizzare efficacemente le operazioni GraphQL.

CloudWatch metrics è uno strumento che fornisce un'ampia gamma di metriche per monitorare le prestazioni e l'utilizzo delle API. Queste metriche si dividono in due categorie principali:

1. Metriche generali sulle API: includono `4XXError` e `5XXError` per tracciare gli errori di client e server, misurare `Latency` i tempi di risposta, monitorare `Requests` le chiamate API totali e tracciare `TokensConsumed` l'utilizzo delle risorse.

2. **Metriche di abbonamento in tempo reale:** queste metriche si concentrano sulle WebSocket connessioni e sulle attività di abbonamento. Includono metriche relative alle richieste di connessione, alle connessioni riuscite, alle registrazioni degli abbonamenti, alla pubblicazione di messaggi e alle connessioni e agli abbonamenti attivi.

La guida introduce anche Enhanced Metrics, che offre dati più granulari sulle prestazioni dei resolver, sulle interazioni con le fonti di dati e sulle singole operazioni GraphQL. Queste metriche forniscono informazioni più approfondite ma comportano costi aggiuntivi.

CloudWatch Logs è uno strumento che abilita le funzionalità di registrazione per GraphQL. APIs I log possono essere impostati su due livelli dell'API:

1. **Registri a livello di richiesta:** acquisiscono informazioni complessive sulle richieste, tra cui intestazioni HTTP, query GraphQL, riepiloghi delle operazioni e registrazioni di sottoscrizioni.
2. **Registri a livello di campo:** forniscono informazioni dettagliate sulle risoluzioni dei singoli campi, incluse le mappature delle richieste e delle risposte e le informazioni di tracciamento per ogni campo.

È possibile configurare la registrazione, interpretare le voci di registro e utilizzare i dati di registro per la risoluzione dei problemi e l'ottimizzazione. AWS AppSync fornisce vari tipi di log che rivelano i dati di esecuzione, analisi, convalida e risoluzione del campo della query.

Installazione e configurazione

Per attivare la registrazione automatica su un'API GraphQL, usa AWS AppSync la console.

1. [Accedi Console di gestione AWS e apri la AppSync console.](#)
2. Nella APIspagina, scegli il nome di un'API GraphQL.
3. Nella home page dell'API, nel pannello di navigazione, scegli Impostazioni.
4. Sotto Logging (Registrazione) segui la procedura riportata di seguito:
 - a. Attiva Abilita registri.
 - b. Per una registrazione dettagliata a livello di richiesta, seleziona la casella di controllo in Includi contenuto dettagliato. (opzionale)
 - c. In Field resolver log level, scegli il livello di registrazione a livello di campo preferito (Nessuno, Error, Info, Debug o All). (opzionale)

- d. In Crea o usa un ruolo esistente, scegli Nuovo ruolo per creare un nuovo AWS Identity and Access Management (IAM) AWS AppSync su cui scrivere CloudWatch i log. Oppure, scegli Ruolo esistente per selezionare l'Amazon Resource Name (ARN) di un ruolo IAM esistente nel tuo AWS account.
5. Scegli Save (Salva).

Configurazione manuale del ruolo IAM

Se scegli di utilizzare un ruolo IAM esistente, il ruolo deve concedere AWS AppSync le autorizzazioni necessarie per scrivere i log. CloudWatch Per configurarlo manualmente, è necessario fornire un ruolo di servizio ARN in modo che AWS AppSync possa assumere il ruolo durante la scrittura dei log.

Nella [console IAM](#), crea una nuova policy con il nome `AWSAppSyncPushToCloudWatchLogsPolicy` che ha la seguente definizione:

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents"
      ],
      "Resource": "*"
    }
  ]
}
```

Quindi, crea un nuovo ruolo con il nome `AWSAppSyncPushToCloudWatchLogsRole` e allega la policy appena creata al ruolo. Modifica la relazione di trust per questo ruolo nel modo seguente:

JSON

```
{
```

```
    "Version": "2012-10-17",
    "Statement": [
      {
        "Effect": "Allow",
        "Principal": {
          "Service": "appsync.amazonaws.com"
        },
        "Action": "sts:AssumeRole"
      }
    ]
  }
}
```

Copia il ruolo ARN e usalo quando configuri la registrazione per un'API AWS AppSync GraphQL.

CloudWatch metriche

Puoi utilizzare le CloudWatch metriche per monitorare e fornire avvisi su eventi specifici che possono causare codici di stato HTTP o latenza. Vengono emesse le seguenti metriche:

Elenco delle metriche

4XXError

Errori derivanti da richieste non valide a causa di una configurazione errata del client. In genere, questi errori si verificano ovunque al di fuori dell'elaborazione GraphQL. Ad esempio, questi errori possono verificarsi quando la richiesta include un payload JSON errato o una query errata, quando il servizio è limitato o quando le impostazioni di autorizzazione non sono configurate correttamente.

Unità: numero. Viene usata la statistica Sum (Somma) per ottenere il numero totale di occorrenze di questi errori.

5XXError

Errori riscontrati durante l'esecuzione di una query GraphQL. Ad esempio, ciò può verificarsi quando si richiama una query per uno schema vuoto o errato. Può verificarsi anche quando l'ID o la AWS regione del pool di utenti di Amazon Cognito non sono validi. In alternativa, ciò potrebbe accadere anche se si AWS AppSync verifica un problema durante l'elaborazione di una richiesta.

Unità: numero. Viene usata la statistica Sum (Somma) per ottenere il numero totale di occorrenze di questi errori.

Latency

Il tempo che intercorre tra il momento in cui AWS AppSync riceve una richiesta da un client e il momento in cui restituisce una risposta al client. Ciò non include la latenza di rete riscontrata per la ricezione di una risposta nei dispositivi finali.

Unità: millisecondi. Viene usata la statistica Average (Media) per valutare le latenze previste.

Requests

Il numero di richieste (interrogazioni+mutazioni) elaborate da tutte le APIs parti del tuo account, per regione.

Unità: numero. Il numero di tutte le richieste elaborate in una particolare regione.

TokensConsumed

I token vengono assegnati in `Requests` base alla quantità di risorse (tempo di elaborazione e memoria utilizzata) consumate da `aRequest`. Di solito, ognuno `Request` consuma un token. Tuttavia, a un `Request` utente che consuma grandi quantità di risorse vengono assegnati token aggiuntivi in base alle esigenze.

Unità: numero. Il numero di token assegnati alle richieste elaborate in una particolare regione.

NetworkBandwidthOutAllowanceExceeded

Note

Nella AWS AppSync console, nella pagina delle impostazioni della cache, l'opzione `Cache Health Metrics` consente di abilitare questa metrica di integrità relativa alla cache.

I pacchetti di rete sono stati interrotti perché la velocità effettiva ha superato il limite di larghezza di banda aggregato. Ciò è utile per diagnosticare i colli di bottiglia in una configurazione di cache. I dati vengono registrati per una particolare API specificando la nella metrica. `API_Id appsyncCacheNetworkBandwidthOutAllowanceExceeded`

Unità: numero. Il numero di pacchetti eliminati dopo aver superato il limite di larghezza di banda per un'API specificata dall'ID.

EngineCPUUtilization

Note

Nella AWS AppSync console, nella pagina delle impostazioni della cache, l'opzione Cache Health Metrics consente di abilitare questa metrica di integrità relativa alla cache.

L'utilizzo della CPU (percentuale) allocato al processo Redis OSS. Ciò è utile per diagnosticare i colli di bottiglia in una configurazione della cache. I dati vengono registrati per una particolare API specificando la metrica. API_Id appsyncCacheEngineCPUUtilization

Unità: Percentuale. La percentuale di CPU attualmente utilizzata dal processo Redis OSS per un'API specificata da ID.

Abbonamenti in tempo reale

Tutte le metriche vengono emesse in un'unica dimensione: Graph. QLAPIId Ciò significa che tutte le metriche sono abbinate all'API GraphQL. IDs Le seguenti metriche sono relative agli abbonamenti GraphQL su pure: WebSockets

Elenco delle metriche

ConnectRequests

Il numero di richieste di WebSocket connessione effettuate a AWS AppSync, inclusi i tentativi riusciti e quelli non riusciti.

Unità: numero. Utilizza la statistica Sum per ottenere il numero totale di richieste di connessione.

ConnectSuccess

Il numero di WebSocket connessioni riuscite a. AWS AppSync È possibile avere connessioni senza sottoscrizioni.

Unità: numero. Viene usata la statistica Sum (Somma) per ottenere il numero totale di occorrenze di connessioni riuscite.

ConnectClientError

Il numero di WebSocket connessioni che sono state rifiutate a AWS AppSync causa di errori sul lato client. Ciò potrebbe implicare che il servizio sia limitato o che le impostazioni di autorizzazione siano configurate in modo errato.

Unità: numero. Viene usata la statistica Sum (Somma) per ottenere il numero totale di occorrenze di errori di connessione lato client.

ConnectServerError

Il numero di errori che hanno avuto origine durante l'elaborazione delle connessioni. AWS AppSync Questo accade in genere quando si verifica un problema imprevisto sul lato server.

Unità: numero. Viene usata la statistica Sum (Somma) per ottenere il numero totale di occorrenze di errori di connessione lato server.

DisconnectSuccess

Il numero di WebSocket disconnessioni riuscite da. AWS AppSync

Unità: numero. Viene usata la statistica Sum (Somma) per ottenere il numero totale di occorrenze di disconnessioni riuscite.

DisconnectClientError

Il numero di errori del client causati dalla WebSocket disconnessione delle AWS AppSync connessioni.

Unità: numero. Viene usata la statistica Sum (Somma) per ottenere il numero totale di occorrenze di errori di disconnessione.

DisconnectServerError

Il numero di errori del server causati dalla disconnessione delle connessioni AWS AppSync .
WebSocket

Unità: numero. Viene usata la statistica Sum (Somma) per ottenere il numero totale di occorrenze di errori di disconnessione.

SubscribeSuccess

Il numero di abbonamenti registrati correttamente su Through. AWS AppSync WebSocket È possibile avere connessioni senza abbonamenti, ma non è possibile avere abbonamenti senza connessioni.

Unità: numero. Viene usata la statistica Sum (Somma) per ottenere il numero totale di occorrenze di sottoscrizioni riuscite.

SubscribeClientError

Il numero di abbonamenti che sono stati rifiutati a AWS AppSync causa di errori sul lato client. Ciò può verificarsi quando un payload JSON non è corretto, il servizio è limitato o le impostazioni di autorizzazione non sono configurate correttamente.

Unità: numero. Viene usata la statistica Sum (Somma) per ottenere il numero totale di occorrenze di errori di sottoscrizione lato client.

SubscribeServerError

Il numero di errori che hanno avuto origine durante l'elaborazione degli abbonamenti. AWS AppSync Questo accade in genere quando si verifica un problema imprevisto sul lato server.

Unità: numero. Viene usata la statistica Sum (Somma) per ottenere il numero totale di occorrenze di errori di sottoscrizione lato server.

UnsubscribeSuccess

Il numero di richieste di annullamento dell'iscrizione che sono state elaborate correttamente.

Unità: numero. Utilizza la statistica Sum per ottenere il numero totale di occorrenze delle richieste di annullamento dell'iscrizione riuscite.

UnsubscribeClientError

Il numero di richieste di annullamento dell'iscrizione che sono state rifiutate a AWS AppSync causa di errori sul lato client.

Unità: numero. Utilizza la statistica Sum per ottenere il numero totale di ricorrenze degli errori di richiesta di annullamento dell'iscrizione sul lato client.

UnsubscribeServerError

Il numero di errori che hanno avuto origine durante l'elaborazione delle richieste di annullamento dell'iscrizione. AWS AppSync Questo accade in genere quando si verifica un problema imprevisto sul lato server.

Unità: numero. Utilizza la statistica Sum per ottenere il numero totale di ricorrenze degli errori di richiesta di annullamento dell'iscrizione sul lato server.

PublishDataMessageSuccess

Numero di messaggi di evento di sottoscrizione pubblicati.

Unità: numero. Viene usata la statistica Sum (Somma) per ottenere il numero totale di messaggi di eventi di sottoscrizione pubblicati.

PublishDataMessageClientError

Numero di messaggi di evento di sottoscrizione che non sono stati pubblicati a causa di errori sul lato client.

Unit: Conta. Viene usata la statistica Sum (Somma) per ottenere il numero totale di occorrenze di errori di eventi di pubblicazione della sottoscrizione lato client.

PublishDataMessageServerError

Il numero di errori che hanno avuto origine AWS AppSync durante la pubblicazione dei messaggi relativi agli eventi di sottoscrizione. Questo accade in genere quando si verifica un problema imprevisto sul lato server.

Unità: numero. Viene usata la statistica Sum (Somma) per ottenere il numero totale di occorrenze di errori di eventi di pubblicazione della sottoscrizione lato server.

PublishDataMessageSize

Dimensione dei messaggi di evento di sottoscrizione pubblicati.

Unità: byte.

ActiveConnections

Il numero di WebSocket connessioni simultanee tra client e AWS AppSync in 1 minuto.

Unità: numero. Viene usata la statistica Sum (Somma) per ottenere il numero totale di connessioni aperte.

ActiveSubscriptions

Numero di sottoscrizioni simultanee dai client in 1 minuto.

Unità: numero. Viene usata la statistica Sum (Somma) per ottenere il numero totale di sottoscrizioni attive.

ConnectionDuration

La quantità di tempo in cui la connessione rimane aperta.

Unità: millisecondi. Viene usata la statistica Average (Media) per valutare la durata della connessione.

OutboundMessages

Il numero di messaggi misurati pubblicati con successo. Un messaggio misurato equivale a 5 kB di dati consegnati.

Unità: numero. Utilizza la statistica Sum per ottenere il numero totale di messaggi misurati pubblicati correttamente.

InboundMessageSuccess

Il numero di messaggi in entrata elaborati correttamente. Ogni tipo di sottoscrizione richiamato da una mutazione genera un messaggio in entrata.

Unità: numero. Utilizza la statistica Sum per ottenere il numero totale di messaggi in entrata elaborati correttamente.

InboundMessageError

Il numero di messaggi in entrata che non sono stati elaborati a causa di richieste API non valide, ad esempio il superamento del limite di 240 kB per il payload dell'abbonamento.

Unità: numero. Utilizza la statistica Sum per ottenere il numero totale di messaggi in entrata con errori di elaborazione relativi all'API.

InboundMessageFailure

Il numero di messaggi in entrata che non sono stati elaborati a causa di errori da AWS AppSync.

Unità: numero. Utilizza la statistica Sum per ottenere il numero totale di messaggi in entrata con errori di elaborazione AWS AppSync correlati.

InboundMessageDelayed

Il numero di messaggi in entrata ritardati. I messaggi in entrata possono subire ritardi quando viene superata la quota di velocità per i messaggi in entrata o la quota per i messaggi in uscita.

Unità: numero. Utilizza la statistica Sum per ottenere il numero totale di messaggi in entrata che hanno subito un ritardo.

InboundMessageDropped

Il numero di messaggi in entrata persi. I messaggi in entrata possono essere eliminati quando viene superata la quota relativa alla tariffa per i messaggi in entrata o la quota per i messaggi in uscita.

Unità: numero. Utilizza la statistica Sum per ottenere il numero totale di messaggi in entrata che sono stati eliminati.

InvalidationSuccess

Il numero di abbonamenti invalidati (annullati) con successo da una mutazione con.
`$extensions.invalidateSubscriptions()`

Unità: numero. Utilizza la statistica Sum per recuperare il numero totale di abbonamenti che sono stati annullati con successo.

InvalidationRequestSuccess

Il numero di richieste di invalidazione elaborate con successo.

Unità: numero. Utilizza la statistica Sum per ottenere il numero totale di richieste di invalidamento elaborate con successo.

InvalidationRequestError

Il numero di richieste di invalidazione che non sono state elaborate a causa di richieste API non valide.

Unità: numero. Utilizza la statistica Sum per ottenere il numero totale di richieste di invalidazione con errori di elaborazione relativi all'API.

InvalidationRequestFailure

Il numero di richieste di invalidazione che non sono state elaborate a causa di errori di. AWS AppSync

Unità: numero. Utilizza la statistica Sum per ottenere il numero totale di richieste di invalidazione con errori di elaborazione correlati. AWS AppSync

InvalidationRequestDropped

Il numero di richieste di invalidazione è diminuito quando è stata superata la quota di richieste di invalidamento.

Unità: numero. Utilizza la statistica Sum per ottenere il numero totale di richieste di invalidazione eliminate.

Confronto tra messaggi in entrata e in uscita

Quando viene eseguita una mutazione, vengono richiamati i campi di sottoscrizione con la direttiva `@aws_subscribe` per quella mutazione. Ogni chiamata di sottoscrizione genera un messaggio in entrata. Ad esempio, se due campi di sottoscrizione specificano la stessa mutazione in `@aws_subscribe`, quando viene chiamata la mutazione vengono generati due messaggi in entrata.

Un messaggio in uscita equivale a 5 kB di dati consegnati ai client. WebSocket Ad esempio, l'invio di 15 kB di dati a 10 client genera 30 messaggi in uscita (15 kB* 10 client/5 kB per messaggio = 30 messaggi).

È possibile richiedere aumenti delle quote per i messaggi in entrata o in uscita. Per ulteriori informazioni, consulta [AWS AppSync endpoint e quote](#) nella guida di riferimento AWS generale e le istruzioni per [richiedere un aumento delle quote](#) nella Service Quotas User Guide.

Metriche migliorate

Le metriche avanzate emettono dati granulari sull'utilizzo e sulle prestazioni delle API, come il conteggio delle AWS AppSync richieste e degli errori, la latenza e gli accessi non riusciti nella cache. Tutti i dati metrici avanzati vengono inviati al tuo CloudWatch account e puoi configurare i tipi di dati che verranno inviati.

Note

Quando si utilizzano metriche avanzate, vengono applicati costi aggiuntivi. Per ulteriori informazioni, consulta i livelli di prezzo di monitoraggio dettagliati nei [CloudWatchprezzi di Amazon](#).

Queste metriche sono disponibili in varie pagine di impostazioni della AWS AppSync console. Nella pagina delle impostazioni API, la sezione Enhanced Metrics consente di abilitare o disabilitare i seguenti elementi:

Comportamento delle metriche dei resolver: queste opzioni controllano il modo in cui vengono raccolte le metriche aggiuntive per i resolver. Puoi scegliere di abilitare le metriche complete dei resolver a richiesta (metriche abilitate per tutti i resolver presenti nelle richieste) o le metriche per

resolver (metriche abilitate solo per i resolver in cui la configurazione è impostata come abilitata). Sono disponibili le seguenti opzioni:

Elenco dei comportamenti delle metriche del resolver

GraphQL errors per resolver (GraphQLError)

Il numero di errori GraphQL che si sono verificati per resolver.

Dimensione metrica:, API_Id Resolver

Unità: numero.

Requests per resolver (Request)

Il numero di chiamate avvenute durante una richiesta. Questo dato viene registrato per ciascun resolver.

Dimensione metrica:, API_Id Resolver

Unità: numero.

Latency per resolver (Latency)

Il tempo necessario per completare una chiamata al resolver. La latenza viene misurata in millisecondi e registrata per ogni resolver.

Dimensione **API_Id** metrica:, Resolver

Unità: millisecondi.

Cache hits per resolver (CacheHit)

Il numero di accessi alla cache durante una richiesta. Questo verrà emesso solo se viene utilizzata una cache. Gli accessi alla cache vengono registrati per ciascun resolver.

Dimensione metrica:, API_Id Resolver

Unità: numero.

Cache misses per resolver (CacheMiss)

Il numero di cache mancanti durante una richiesta. Questo verrà emesso solo se viene utilizzata una cache. Gli errori di cache vengono registrati per ciascun resolver.

Dimensione metrica:, API_Id Resolver

Unità: numero.

Comportamento delle metriche delle origini dati: queste opzioni controllano il modo in cui vengono raccolte le metriche aggiuntive per le fonti di dati. Puoi scegliere di abilitare le metriche complete dell'origine dati della richiesta (metriche abilitate per tutte le origini dati nelle richieste) o le metriche per origine dati (le metriche sono abilitate solo per le fonti di dati in cui la configurazione è impostata su abilitata). Sono disponibili le seguenti opzioni:

Elenco dei comportamenti delle metriche delle fonti di dati

Requests per data source (Request)

Il numero di chiamate avvenute durante una richiesta. Le richieste vengono registrate in base alla fonte dei dati. Se le richieste complete sono abilitate, ogni fonte di dati avrà la propria immissione. CloudWatch

Dimensione metrica: API_Id, Datasource

Unità: numero.

Latency per data source (Latency)

Il tempo necessario per completare una chiamata all'origine dati. La latenza viene registrata in base alla fonte dei dati.

Dimensione metrica: API_Id Datasource

Unità: millisecondi.

Errors per data source (GraphQLError)

Il numero di errori che si sono verificati durante una chiamata all'origine dati.

Dimensione metrica: API_Id Datasource

Unità: numero.

Metriche operative: abilita le metriche a livello operativo GraphQL.

Elenco dei comportamenti delle metriche operative

Requests per operation (Request)

Il numero di volte in cui è stata chiamata un'operazione GraphQL specificata.

Dimensione metrica:, API_Id Operation

Unità: numero.

GraphQL errors per operation (GraphQLError)

Il numero di errori GraphQL che si sono verificati durante un'operazione GraphQL specificata.

Dimensione metrica:, API_Id Operation

Unità: numero.

CloudWatch registri

È possibile configurare due tipi di logging per qualsiasi API GraphQL nuova o esistente: a livello di richiesta e a livello di campo.

Registri a livello di richiesta

Quando è configurata la registrazione a livello di richiesta (include contenuti dettagliati), vengono registrate le seguenti informazioni:

- Il numero di token consumati
- Intestazioni HTTP di richiesta e risposta
- La query GraphQL in esecuzione nella richiesta
- Il riepilogo generale dell'operazione
- Sottoscrizioni GraphQL nuove ed esistenti registrate

Registri a livello di campo

Quando è configurata la registrazione a livello di campo, vengono registrate le seguenti informazioni:

- Mappatura delle richieste generata con origine e argomenti per ogni campo
- La mappatura delle risposte trasformata per ogni campo, che include i dati risultanti dalla risoluzione di quel campo
- Informazioni di traccia per ogni campo

Se si attiva la registrazione, AWS AppSync gestisce i registri. CloudWatch Il processo include la creazione di gruppi e flussi di log e la segnalazione di tali log ai flussi di log.

Quando attivi la registrazione su un'API GraphQL e fai richieste AWS AppSync, crea un gruppo di log e flussi di log all'interno del gruppo di log. Al gruppo di log viene assegnato un nome nel formato `/aws/appsync/apis/{graphql_api_id}`. In ogni gruppo di log, i log vengono ulteriormente suddivisi in flussi di log. Questi sono ordinati in base al valore Last Event Time (Ora ultimo evento) quando vengono segnalati i dati registrati.

Ogni evento di registro è contrassegnato con il codice `x-amzn-requestid` - di quella richiesta. Questo ti aiuta a filtrare gli eventi di registro CloudWatch per ottenere tutte le informazioni registrate su quella richiesta. Puoi ottenerlo `RequestId` dalle intestazioni di risposta di ogni richiesta AWS AppSync GraphQL.

Il logging a livello di campo è configurato con i livelli di log seguenti:

- Nessuno: non viene acquisito alcun registro a livello di campo.
- Errore: registra le seguenti informazioni solo per i campi che rientrano nella categoria di errore:
 - Sezione dell'errore nella risposta del server
 - Errori a livello di campo
 - Le request/response funzioni generate che sono state risolte per i campi di errore
- Informazioni: registra le seguenti informazioni solo per i campi che si trovano nelle categorie informazioni ed errori:
 - Messaggi a livello di informazioni
 - I messaggi utente inviati tramite `$util.log.info console.log`
 - I registri di tracciamento e mappatura a livello di campo non vengono visualizzati.
 - Se la registrazione a livello di campo è impostata su INFO o superiore con contenuto verboso incluso, aggiunge i messaggi di registrazione del modello di mappatura trasformato. AWS AppSync Contrerà tutte le informazioni aggiunte al modello di mappatura trasformato o l'output del resolver o del JavaScript codice eseguito dalla funzione, e non deve essere utilizzato se si prevede di inviare informazioni riservate, come password o intestazioni di autorizzazione, a fonti di dati a valle e non si desidera che tali informazioni vengano inserite nei log.
- Debug: registra le seguenti informazioni solo per i campi che si trovano nelle categorie debug, info ed error:
 - Messaggi a livello di debug
 - I messaggi utente inviati tramite `$util.log.info,,` e `$util.log.debug console.log console.debug`

- I registri di tracciamento e mappatura a livello di campo non vengono visualizzati.
- Tutti: registra le seguenti informazioni per tutti i campi della query:
 - Informazioni di traccia a livello di campo
 - Le request/response funzioni generate che sono state risolte per ogni campo

Vantaggi del monitoraggio

Puoi usare il logging e i parametri per identificare e ottimizzare le query GraphQL, oltre che per risolvere i relativi problemi. Puoi ad esempio eseguire il debug dei problemi di latenza usando le informazioni di traccia registrate per ogni campo nella query. Per dimostrare questo concetto, supponiamo di usare uno o più resolver annidati in una query GraphQL. Un esempio di operazione sul campo in CloudWatch Logs potrebbe essere simile alla seguente:

```
{
  "path": [
    "singlePost",
    "authors",
    0,
    "name"
  ],
  "parentType": "Post",
  "returnType": "String!",
  "fieldName": "name",
  "startOffset": 416563350,
  "duration": 11247
}
```

Ciò potrebbe corrispondere a uno schema GraphQL analogo a quanto segue:

```
type Post {
  id: ID!
  name: String!
  authors: [Author]
}

type Author {
  id: ID!
  name: String!
}
```



```
type Query {  
  singlePost(id:ID!): Post  
}
```

Nei risultati del registro precedenti, path mostra un singolo elemento nei dati restituiti dall'esecuzione di una query denominata. `singlePost()` In questo esempio, rappresenta il campo del nome nel primo indice (0). Lo `startOffset` fornisce un offset dall'inizio dell'operazione di interrogazione GraphQL. La durata è il tempo totale per risolvere il campo. Questi valori possono essere utili per risolvere un problema a causa di cui i dati provenienti da un'origine dati specifica vengono eseguiti più lentamente del previsto oppure un campo specifico rallenta l'intera query. Ad esempio, puoi scegliere di aumentare il throughput assegnato per una tabella Amazon DynamoDB o rimuovere un campo specifico da una query che causa un cattivo funzionamento dell'operazione complessiva.

A partire dall'8 maggio 2019, AWS AppSync genera eventi di registro in formato JSON completamente strutturato. Questo può aiutarti a utilizzare servizi di analisi dei log come CloudWatch Logs Insights e Amazon OpenSearch Service per comprendere le prestazioni delle tue richieste GraphQL e le caratteristiche di utilizzo dei campi dello schema. Ad esempio, è possibile identificare in modo semplice i resolver con latenze di grandi dimensioni, possibile causa principale di un problema prestazionale. Inoltre, ora è possibile identificare i campi utilizzati più e meno frequentemente all'interno dello schema e valutare l'impatto di definire come obsoleti i campi GraphQL.

Rilevamento dei conflitti e registrazione della sincronizzazione

Se un' AWS AppSync API ha configurato la registrazione su CloudWatch Logs con il livello di registro Field resolver impostato su All, invia informazioni sul rilevamento e AWS AppSync la risoluzione dei conflitti al gruppo di log. Ciò fornisce informazioni dettagliate su come l'API ha risposto a un conflitto. AWS AppSync Per aiutarti a interpretare la risposta, nei log vengono fornite le seguenti informazioni:

Elenco delle metriche

`conflictType`

Indica nei dettagli se si è verificato un conflitto a causa di una mancata corrispondenza della versione o della condizione fornita dal cliente.

`conflictHandlerConfigured`

Indica il gestore di conflitti configurato nel resolver al momento della richiesta.

`message`

Fornisce informazioni su come il conflitto è stato rilevato e risolto.

syncAttempt

Il numero di tentativi che il server ha effettuato per sincronizzare i dati prima di rifiutare la richiesta.
data

Se il gestore dei conflitti configurato è Automerge, questo campo viene compilato per mostrare la decisione Automerge presa per ogni campo. Le operazioni fornite possono essere:

- **RIFIUTATO**: quando Automerge rifiuta il valore del campo in entrata a favore del valore nel server.
- **AGGIUNTO** - Quando viene Automerge aggiunto al campo in entrata a causa dell'assenza di un valore preesistente nel server.
- **AGGIUNTO**: When Automerge aggiunge i valori in entrata ai valori dell'elenco esistente nel server.
- **MERGED** - Automerge When unisce i valori in entrata ai valori del Set esistente nel server.

Utilizzo del conteggio dei token per ottimizzare le richieste

Alle richieste che consumano meno o pari a 1.500 KB al secondo di memoria e tempo di vCPU viene allocato un token. Le richieste con un consumo di risorse superiore a 1.500 KB al secondo ricevono token aggiuntivi. Ad esempio, se una richiesta consuma 3.350 KB al secondo, AWS AppSync alloca tre token (arrotondati al valore intero successivo) alla richiesta. Per impostazione predefinita, AWS AppSync alloca un massimo di 5.000 o 10.000 token di richiesta al secondo APIs nel tuo account, a seconda della regione in cui viene distribuito. AWS Se APIs ognuno di voi utilizza una media di due token al secondo, sarete limitati a 2.500 o 5.000 richieste al secondo, rispettivamente. Se hai bisogno di più token al secondo rispetto all'importo assegnato, puoi inviare una richiesta per aumentare la quota predefinita per la frequenza di token di richiesta. Per ulteriori informazioni, consulta [AWS AppSync Endpoint e quote](#) nella Riferimenti generali di AWS guida e [Richiesta di un aumento delle quote](#) nella Service Quotas User Guide.

Un numero elevato di token per richiesta potrebbe indicare che esiste l'opportunità di ottimizzare le richieste e migliorare le prestazioni dell'API. I fattori che possono aumentare il numero di token per richiesta includono:

- Le dimensioni e la complessità dello schema GraphQL.
- La complessità dei modelli di mappatura delle richieste e delle risposte.
- Il numero di chiamate del resolver per richiesta.

- La quantità di dati restituita dai resolver.
- La latenza delle fonti di dati downstream.
- Progettazioni di schemi e query che richiedono chiamate successive all'origine dati (al contrario delle chiamate parallele o in batch).
- Configurazione della registrazione, in particolare contenuti di log dettagliati e a livello di campo.

Note

Oltre alle AWS AppSync metriche e ai log, i client possono accedere al numero di token utilizzati in una richiesta tramite l'intestazione di risposta. `x-amzn-appsync-TokensConsumed`

Limiti di dimensione del registro

Per impostazione predefinita, se la registrazione è stata abilitata, AWS AppSync invierà fino a 1 MB di log per richiesta. I log che superano questa dimensione verranno troncati. Per ridurre le dimensioni dei log, scegli il livello di registrazione per i ERROR log a livello di campo e disabilita la registrazione, oppure disabilita VERBOSE completamente i log a livello di campo se non necessario. In alternativa al livello di ALL log, puoi utilizzare Enhanced Metrics per ottenere metriche su resolver, fonti di dati o operazioni GraphQL specifici, oppure utilizzare le utilità di registrazione fornite da per registrare solo le informazioni necessarie. AppSync

Riferimento al tipo di registro

RequestSummary

- `requestId`: identificatore univoco per la richiesta.
- `graphqlAPIId`: ID dell'API GraphQL che effettua la richiesta.
- `statusCode`: risposta al codice di stato HTTP.
- `latenza`: End-to-end latenza della richiesta, in nanosecondi, come numero intero.

```
{
  "logType": "RequestSummary",
  "requestId": "dbe87af3-c114-4b32-ae79-8af11f3f96f1",
  "graphqlAPIId": "pmo28inf75eepg63qxq4ekoeg4",
```

```
    "statusCode": 200,  
    "latency": 242000000  
}
```

ExecutionSummary

- **requestId**: identificatore univoco per la richiesta.
- **graphQLAPIId**: ID dell'API GraphQL che effettua la richiesta.
- **StartTime**: il timestamp di inizio dell'elaborazione GraphQL per la richiesta, in formato RFC 3339.
- **EndTime**: il timestamp di fine dell'elaborazione GraphQL per la richiesta, in formato RFC 3339.
- **durata**: Il tempo di elaborazione GraphQL totale trascorso, in nanosecondi, come numero intero.
- **versione**: La versione dello schema di ExecutionSummary
- **parsing**:
 - **startOffset**: l'offset iniziale per l'analisi, in nanosecondi, rispetto all'invocazione, come numero intero.
 - **duration**: il tempo impiegato per l'analisi, in nanosecondi, come numero intero.
- **validation**:
 - **startOffset**: l'offset iniziale per la convalida, in nanosecondi, relativo all'invocazione, come numero intero.
 - **duration**: il tempo impiegato per eseguire la convalida, in nanosecondi, come numero intero.

```
{  
  "duration": 217406145,  
  "logType": "ExecutionSummary",  
  "requestId": "dbe87af3-c114-4b32-ae79-8af11f3f96f1",  
  "startTime": "2019-01-01T06:06:18.956Z",  
  "endTime": "2019-01-01T06:06:19.174Z",  
  "parsing": {  
    "startOffset": 49033,  
    "duration": 34784  
  },  
  "version": 1,  
  "validation": {  
    "startOffset": 129048,  
    "duration": 69126  
  },  
  "graphQLAPIId": "pmo28inf75eepg63qxq4ekoeg4"
```

```
}
```

Tracciamento

- **requestId**: identificatore univoco per la richiesta.
- **graphqlAPIId**: ID dell'API GraphQL che effettua la richiesta.
- **startOffset**: l'offset iniziale per la risoluzione del campo, in nanosecondi, rispetto all'invocazione, come numero intero.
- **duration**: il tempo impiegato per risolvere il campo, in nanosecondi, come numero intero.
- **fieldName**: il nome del campo in fase di risoluzione.
- **parentType**: il tipo padre del campo in fase di risoluzione.
- **returnType**: il tipo restituito del campo in fase di risoluzione.
- **path**: un elenco di segmenti di percorso, che inizia dalla radice della risposta e termina con il campo in fase di risoluzione.
- **resolverArn**: l'ARN del resolver utilizzato per la risoluzione del campo. Potrebbe non essere presente nei campi nidificati.

```
{
  "duration": 216820346,
  "logType": "Tracing",
  "path": [
    "putItem"
  ],
  "fieldName": "putItem",
  "startOffset": 178156,
  "resolverArn": "arn:aws:appsync:us-east-1:111111111111:apis/
pmo28inf75eepg63qxq4ekoeg4/types/Mutation/fields/putItem",
  "requestId": "dbe87af3-c114-4b32-ae79-8af11f3f96f1",
  "parentType": "Mutation",
  "returnType": "Item",
  "graphqlAPIId": "pmo28inf75eepg63qxq4ekoeg4"
}
```

Analisi CloudWatch dei log con Logs Insights

Di seguito sono elencati alcuni esempi di query che è possibile eseguire per ottenere informazioni dettagliate sulle prestazioni e lo stato delle operazioni GraphQL. Questi esempi sono disponibili come

query di esempio nella console Logs Insights. CloudWatch Nella [CloudWatchconsole](#), scegli Logs Insights, seleziona il gruppo di AWS AppSync log per la tua API GraphQL, quindi AWS AppSync scegli le query in Query di esempio.

La seguente query restituisce le prime 10 richieste GraphQL con il numero massimo di token consumati:

```
filter @message like "Tokens Consumed"
| parse @message "* Tokens Consumed: *" as requestId, tokens
| sort tokens desc
| display requestId, tokens
| limit 10
```

La query seguente restituisce i primi 10 resolver con latenza massima:

```
fields resolverArn, duration
| filter logType = "Tracing"
| limit 10
| sort duration desc
```

La query seguente restituisce i resolver richiamati più di frequente:

```
fields ispresent(resolverArn) as isRes
| stats count() as invocationCount by resolverArn
| filter isRes and logType = "Tracing"
| limit 10
| sort invocationCount desc
```

La query seguente restituisce i resolver con il maggior numero di errori nei modelli di mappatura:

```
fields ispresent(resolverArn) as isRes
| stats count() as errorCount by resolverArn, logType
| filter isRes and (logType = "RequestMapping" or logType = "ResponseMapping") and
fieldInError
| limit 10
| sort errorCount desc
```

La query seguente restituisce le statistiche di latenza dei resolver:

```
fields ispresent(resolverArn) as isRes
| stats min(duration), max(duration), avg(duration) as avg_dur by resolverArn
```

```
| filter isRes and logType = "Tracing"  
| limit 10  
| sort avg_dur desc
```

La query seguente restituisce le statistiche di latenza dei campi:

```
stats min(duration), max(duration), avg(duration) as avg_dur  
by concat(parentType, '/', fieldName) as fieldKey  
| filter logType = "Tracing"  
| limit 10  
| sort avg_dur desc
```

I risultati delle query di CloudWatch Logs Insights possono essere esportati nei dashboard. CloudWatch

Analizza i tuoi log con Service OpenSearch

Puoi cercare, analizzare e visualizzare i tuoi AWS AppSync log con Amazon OpenSearch Service per identificare i punti deboli delle prestazioni e le cause principali dei problemi operativi. Puoi identificare i resolver con la latenza massima e gli errori. Inoltre, puoi utilizzare OpenSearch Dashboards per creare dashboard con visualizzazioni potenti. OpenSearch Dashboards è uno strumento open source di visualizzazione ed esplorazione dei dati disponibile in Service. OpenSearch Utilizzando OpenSearch le dashboard, puoi monitorare continuamente le prestazioni e lo stato delle tue operazioni GraphQL. Ad esempio, puoi creare dashboard per visualizzare la latenza P90 delle tue richieste GraphQL e approfondire le latenze P90 di ciascun resolver.

Quando usi OpenSearch Service, usa «cwl*» come modello di filtro per cercare gli indici.

OpenSearch OpenSearch Il servizio indicizza i log trasmessi in streaming da CloudWatch Logs con il prefisso «cwl-». Per differenziare i log AWS AppSync delle API dagli altri CloudWatch registri inviati al OpenSearch Servizio, consigliamo di aggiungere un'espressione di filtro aggiuntiva di alla ricerca. `graphqlAPIID.keyword=YourGraphQLAPIID`

Migrazione del formato di registro

Gli eventi di registro AWS AppSync generati sono formattati principalmente come JSON completamente strutturato. Tuttavia, alcuni messaggi di diagnostica e di elaborazione intermedia possono essere emessi in un formato non strutturato. [Se è necessario migrare i log non strutturati in un formato JSON completamente strutturato, è possibile utilizzare uno script disponibile nell'esempio. \[GitHub\]\(#\)](#)

Puoi anche utilizzare [i filtri metrici](#) CloudWatch per trasformare i dati di registro in CloudWatch metriche numeriche, in modo da poterli rappresentare graficamente o impostare un allarme su di essi.

Utilizzo AWS X-Ray per tracciare le richieste in AWS AppSync

È possibile utilizzarlo [AWS X-Ray](#) per tracciare le richieste man mano che vengono eseguite in AWS AppSync. È possibile utilizzare X-Ray con AWS AppSync in tutte le AWS regioni in cui X-Ray è disponibile. X-Ray offre una panoramica dettagliata di un'intera richiesta GraphQL. Ciò consente di analizzare le latenze nei resolver e nelle fonti di dati sottostanti APIs e nelle relative fonti di dati. È possibile utilizzare una mappa dei servizi X-Ray per visualizzare la latenza di una richiesta, inclusi tutti AWS i servizi integrati con X-Ray. È inoltre possibile configurare le regole di campionamento per indicare a X-Ray quali richieste registrare e a quali frequenze di campionamento, in base ai criteri specificati.

Per ulteriori informazioni sul campionamento in X-Ray, [vedere Configurazione delle regole di campionamento](#) nella console. AWS X-Ray

Installazione e configurazione

È possibile abilitare il tracciamento X-Ray per un'API GraphQL tramite la console. AWS AppSync

1. Accedi alla console. AWS AppSync
2. Nel riquadro di navigazione scegliere Settings (Impostazioni).
3. In X-Ray, attivare Enable X-Ray (Abilita X-Ray).
4. Scegli Save (Salva). Il tracciamento X-Ray è ora abilitato per la tua API.

Se utilizzi AWS CLI o AWS CloudFormation, puoi anche abilitare il tracciamento X-Ray quando crei una nuova AWS AppSync API o aggiorni un' AWS AppSync API esistente, impostando la `xrayEnabled` proprietà su `true`

Quando il tracciamento X-Ray è abilitato per un' AWS AppSync API, nell'account viene creato automaticamente un [ruolo AWS Identity and Access Management collegato al servizio](#) con le autorizzazioni appropriate. Ciò consente di AWS AppSync inviare tracce a X-Ray in modo sicuro.

Tracciamento delle API con X-Ray

Campionamento

Utilizzando le regole di campionamento è possibile controllare la quantità di dati da registrare in AWS AppSync e modificare immediatamente il comportamento del campionamento senza dover cambiare o ridistribuire il codice. Ad esempio, questa regola esegue il campionamento delle richieste all'API GraphQL con l'ID API 3n572shhcpfokwhdnq1ogu59v6.

- Nome regola - test-sample
- Priorità - 10
- Dimensioni riserva - 10
- Tasso fisso - 10
- Nome servizio - *
- Tipo di servizio - AWS::AppSync::GraphQLAPI
- Metodo HTTP - *
- Risorsa ARN - arn:aws:appsync:us-west-2:123456789012:apis/3n572shhcpfokwhdnq1ogu59v6
- Host - *

Informazioni sulle tracce

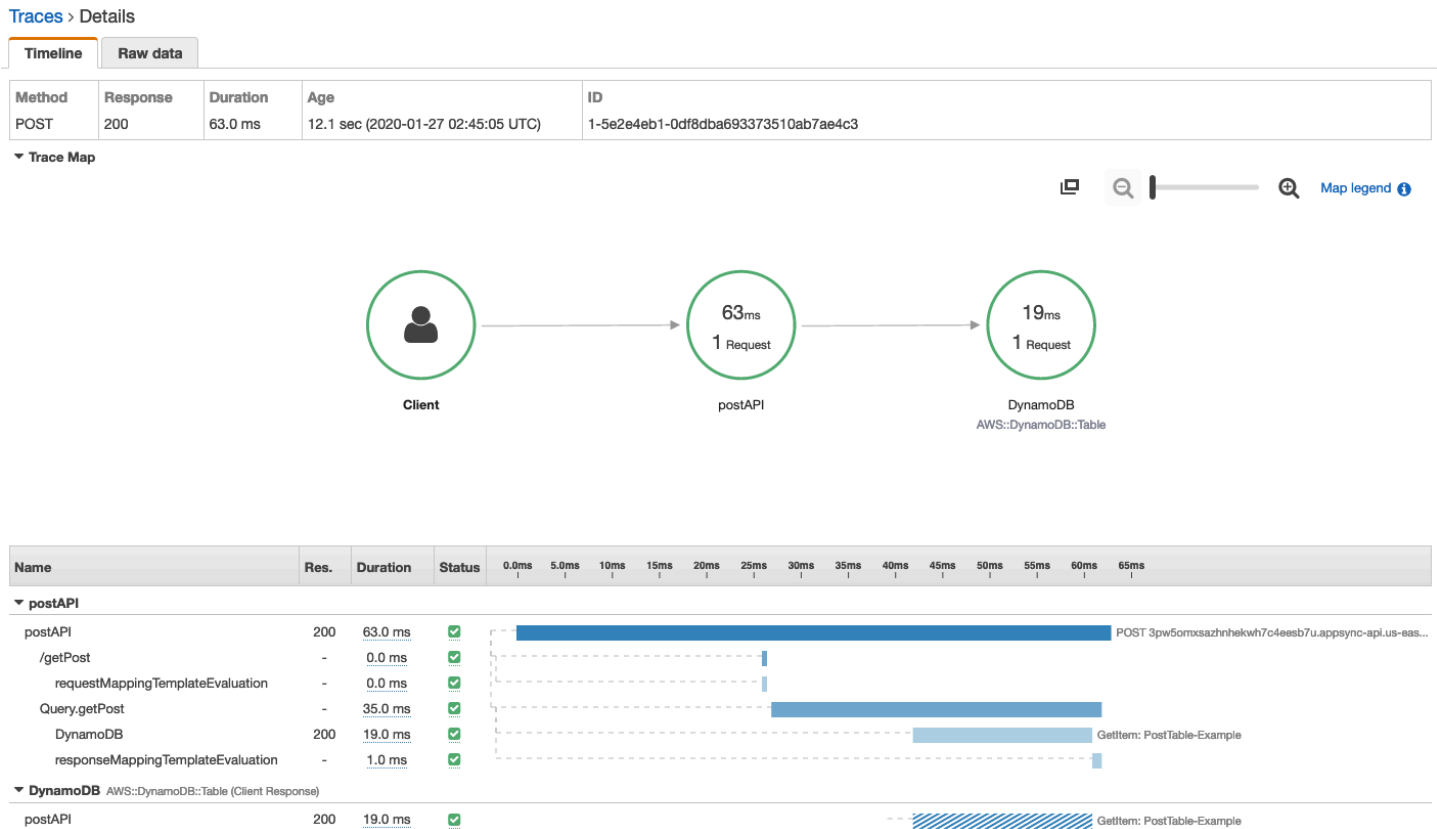
Quando abiliti il tracciamento X-Ray per la tua API GraphQL, puoi utilizzare la pagina dei dettagli della traccia X-Ray per esaminare informazioni dettagliate sulla latenza relative alle richieste fatte alla tua API. Nell'esempio seguente viene illustrata la visualizzazione della traccia insieme alla mappa del servizio per questa specifica richiesta. La richiesta è stata effettuata a un'API chiamata postAPI con un tipo Post, i cui dati sono contenuti in una tabella Amazon DynamoDB chiamata PostTable-Example

L'immagine di traccia seguente corrisponde alla seguente query GraphQL:

```
query getPost {  
  getPost(id: "1") {  
    id  
    title  
  }  
}
```

```
}
}
```

Il resolver per la `getPost` query utilizza l'origine dati DynamoDB sottostante. La seguente visualizzazione di traccia mostra la chiamata a DynamoDB, nonché le latenze di varie parti dell'esecuzione della query:



- Nell'immagine precedente, `/getPost` rappresenta il percorso completo dell'elemento che viene risolto. In questo caso, poiché `getPost` è un campo sul tipo di Query radice, viene visualizzato direttamente dopo la radice del percorso.
- `requestMappingTemplateEvaluation` rappresenta il tempo trascorso da AWS AppSync per la valutazione del modello di mapping della richiesta per questo elemento nella query.
- `Query.getPost` rappresenta un tipo e un campo (nel formato `Type.field`). Può contenere più sottosegmenti, a seconda della struttura dell'API e della richiesta tracciata.
- `DynamoDB` rappresenta l'origine dati associata a questo resolver. Contiene la latenza della chiamata di rete a DynamoDB per risolvere il campo.
- `responseMappingTemplateEvaluation` rappresenta il tempo trascorso da AWS AppSync per la valutazione del modello di mapping della risposta per questo elemento nella query.

Quando si visualizzano le tracce in X-Ray, è possibile ottenere ulteriori informazioni contestuali e sui metadati sui sottosegmenti del AWS AppSync segmento scegliendo i sottosegmenti ed esplorando la vista dettagliata.

[Per alcune interrogazioni complesse o complesse, tenete presente che il segmento inviato a X-Ray da AWS AppSync può essere più grande della dimensione massima consentita per i documenti di segmento, come definito in \[Segment Documents.AWS X-Ray\]\(#\)](#) X-Ray non visualizza i segmenti che superano il limite.

Registrazione delle chiamate AWS AppSync API utilizzando AWS CloudTrail

AWS AppSync è integrato con AWS CloudTrail, un servizio che fornisce una registrazione delle azioni intraprese da un utente, ruolo o AWS servizio in AWS AppSync. CloudTrail acquisisce tutte le chiamate API AWS AppSync come eventi. Le chiamate acquisite includono le chiamate dalla AWS AppSync console e le chiamate in codice a. AWS AppSync APIs È possibile utilizzare le informazioni raccolte da CloudTrail per determinare la richiesta a cui è stata effettuata AWS AppSync, l'indirizzo IP del richiedente, chi ha effettuato la richiesta, quando è stata effettuata la richiesta e ulteriori dettagli.

Puoi creare un percorso per consentire la distribuzione continua di CloudTrail eventi a un bucket Amazon Simple Storage Service (Amazon S3), inclusi gli eventi per. AWS AppSync Se non configuri un percorso, puoi comunque visualizzare gli eventi più recenti nella CloudTrail console.

Per ulteriori informazioni in merito CloudTrail, consulta la [Guida AWS CloudTrail per l'utente](#).

AWS AppSync informazioni in CloudTrail

CloudTrail è abilitato sul tuo AWS account al momento della creazione dell'account. Nella CloudTrail console, in Cronologia eventi, puoi visualizzare, cercare e scaricare gli eventi recenti del tuo AWS account. Per ulteriori informazioni, consulta [Visualizzazione degli eventi con la cronologia degli CloudTrail eventi](#) nella Guida AWS CloudTrail per l'utente.

Per una registrazione continua degli eventi nel tuo AWS account, inclusi gli eventi per AWS AppSync, crea un percorso. Per impostazione predefinita, quando si crea un trail nella console, il trail sarà valido in tutte le Regioni AWS . Il trail registra gli eventi di tutte le regioni della AWS partizione e consegna i file di log al bucket Amazon S3 specificato. Inoltre, puoi configurare altri AWS servizi per analizzare ulteriormente e agire in base ai dati sugli eventi raccolti nei log. CloudTrail Per ulteriori informazioni, consultare gli argomenti seguenti nella Guida per l'utente di AWS CloudTrail :

- [Creazione di un percorso per il tuo account AWS](#)
- [AWS Integrazioni di servizi con log CloudTrail](#)
- [Configurazione delle notifiche Amazon SNS per CloudTrail](#)
- [Ricezione di file di CloudTrail registro da più regioni](#)
- [Ricezione di file di CloudTrail registro da più account](#)

CloudTrail registra tutte le operazioni AWS AppSync API. Ad esempio, chiama e `ListResolvers` APIs genera voci nei file di CloudTrail registro. `CreateGraphQLApi` `CreateDataSource` Queste e altre operazioni sono documentate nell'[AWS AppSync API Reference](#).

Ogni evento o voce di log contiene informazioni sull'utente che ha generato la richiesta. Le informazioni di identità consentono di stabilire:

- Se la richiesta è stata effettuata con credenziali utente root o AWS Identity and Access Management (IAM).
- Se la richiesta è stata effettuata con le credenziali di sicurezza temporanee per un ruolo o un utente federato.
- Se la richiesta è stata effettuata da un altro AWS servizio.

Per ulteriori informazioni, consulta [CloudTrail UserIdentity Element nella Guida](#) per l'AWS CloudTrail utente.

AWS AppSync eventi di dati in CloudTrail

Gli [eventi di dati](#) forniscono informazioni sulle operazioni delle risorse eseguite su o in una risorsa (ad esempio, lettura o scrittura su un oggetto Amazon S3). Queste operazioni sono definite anche operazioni del piano dei dati. Gli eventi di dati sono spesso attività che interessano volumi elevati di dati. Per impostazione predefinita, CloudTrail non registra gli eventi relativi ai dati. La cronologia CloudTrail degli eventi non registra gli eventi relativi ai dati.

Per gli eventi di dati sono previsti costi aggiuntivi. Per ulteriori informazioni sui CloudTrail prezzi, consulta la sezione [AWS CloudTrail Prezzi](#).

Puoi registrare gli eventi relativi ai dati per il tipo di `AWS::AppSync::GraphQLApi` risorsa utilizzando la CloudTrail console o le operazioni CloudTrail API (tra cui operazioni di query, mutazione e sottoscrizione, operazioni di connessione all'endpoint in tempo reale, ma non i messaggi inviati tramite l' WebSocket endpoint in tempo reale). AWS CLI Per ulteriori informazioni

su come registrare gli eventi relativi ai dati, consulta [Registrazione di eventi relativi ai dati con Console di gestione AWS](#) e [Registrazione di eventi di dati con AWS Command Line Interface](#) nella Guida per l'utente AWS CloudTrail .

La tabella seguente elenca il tipo di AWS AppSync risorsa per cui è possibile registrare gli eventi relativi ai dati. La colonna Data event type (console) mostra il valore da scegliere dall'elenco Data event type (console) nella CloudTrail console. La colonna del valore resources.type mostra il resources.type valore da specificare durante la configurazione dei selettori di eventi avanzati utilizzando o. AWS CLI CloudTrail APIs La CloudTrail colonna Dati APIs registrati mostra le chiamate API registrate per il tipo di risorsa. CloudTrail

Tipo di evento di dati (console)	valore resources.type	Dati registrati APIs su CloudTrail
AppSync GraphQL	AWS::AppSync::GraphQLApi	GraphQL

È possibile configurare selettori di eventi avanzati per filtrare i campi eventName, readOnly e resources.ARN per registrare solo gli eventi importanti per l'utente. Per ulteriori informazioni su questi campi, consulta [AdvancedFieldSelector](#) in Riferimento API AWS CloudTrail .

```
[
  {
    "name": "Only 1 AppSync API",
    "fieldSelectors": [
      {
        "field": "eventCategory",
        "equals": [
          "Data"
        ]
      },
      {
        "field": "resources.type",
        "equals": [
          "AWS::AppSync::GraphQLApi"
        ]
      },
      {
        "field": "resources.ARN",
        "equals": [
```

```

        "arn:aws:appsync:us-east-1:111122223333:apis/YourGraphQLApiId"
    ]
}
]
}
]

```

Comprendere le AWS AppSync voci dei file di registro

CloudTrail fornisce gli eventi come file di registro che contengono una o più voci di registro. Un evento rappresenta una singola richiesta proveniente da qualsiasi fonte e include informazioni sull'operazione richiesta, la data e l'ora dell'operazione, i parametri della richiesta e così via. Poiché questi file di registro non sono una traccia ordinata delle chiamate API pubbliche, non vengono visualizzati in un ordine specifico.

Note

`requestIDNon` è un ID univoco autorevole per i log emessi da AWS AppSync.
`requestID` può essere sovrascritto dal client. Pertanto, è necessario prestare attenzione quando si prendono decisioni sulla base di queste informazioni.

L'esempio seguente di voce di CloudTrail registro illustra l'`CreateApiKey` operazione.

```

{
  "Records": [{
    "eventVersion": "1.05",
    "userIdentity": {
      "type": "IAMUser",
      "principalId": "A1B2C3D4E5F6G7EXAMPLE",
      "arn": "arn:aws:iam::111122223333:user/Alice",
      "accountId": "111122223333",
      "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
      "userName": "diego_ramirez"
    },
    "eventTime": "2018-01-31T21:49:09Z",
    "eventSource": "appsync.amazonaws.com",
    "eventName": "CreateApiKey",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "192.2.0.1",
    "userAgent": "aws-cli/1.11.72 Python/2.7.11 Darwin/16.7.0 botocore/1.5.35",
  ]
}

```

```

    "requestParameters": {
      "apiId": "a1b2c3d4e5f6g7h8i9jexample"
    },
    "responseElements": {
      "apiKey": {
        "id": "****",
        "expires": 1518037200000
      }
    },
    "requestID": "99999999-9999-9999-9999-999999999999",
    "eventID": "99999999-9999-9999-9999-999999999999",
    "readOnly": false,
    "eventType": "AwsApiCall",
    "recipientAccountId": "111122223333"
  }
]
}

```

L'esempio seguente di voce di CloudTrail registro illustra l'ListApiKeysoperazione.

```

{
  "Records": [{
    "eventVersion": "1.05",
    "userIdentity": {
      "type": "IAMUser",
      "principalId": "A1B2C3D4E5F6G7EXAMPLE",
      "arn": "arn:aws:iam::111122223333:user/diego_ramirez",
      "accountId": "111122223333",
      "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
      "userName": "diego_ramirez"
    },
    "eventTime": "2018-01-31T21:49:09Z",
    "eventSource": "appsync.amazonaws.com",
    "eventName": "ListApiKeys",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "192.2.0.1",
    "userAgent": "aws-cli/1.11.72 Python/2.7.11 Darwin/16.7.0 botocore/1.5.35",
    "requestParameters": {
      "apiId": "a1b2c3d4e5f6g7h8i9jexample"
    },
    "responseElements": {
      "apiKeys": [
        {

```

```

        "id": "****",
        "expires": 1517954400000
      },
      {
        "id": "****",
        "expires": 1518037200000
      },
    ]
  },
  "requestID": "99999999-9999-9999-9999-999999999999",
  "eventID": "99999999-9999-9999-9999-999999999999",
  "readOnly": false,
  "eventType": "AwsApiCall",
  "recipientAccountId": "111122223333"
}
]
}

```

L'esempio seguente di voce di CloudTrail registro illustra l'DeleteApiKeyoperazione.

```

{
  "Records": [{
    "eventVersion": "1.05",
    "userIdentity": {
      "type": "IAMUser",
      "principalId": "A1B2C3D4E5F6G7EXAMPLE",
      "arn": "arn:aws:iam::111122223333:user/diego_ramirez",
      "accountId": "111122223333",
      "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
      "userName": "diego_ramirez"
    },
    "eventTime": "2018-01-31T21:49:09Z",
    "eventSource": "appsync.amazonaws.com",
    "eventName": "DeleteApiKey",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "192.2.0.1",
    "userAgent": "aws-cli/1.11.72 Python/2.7.11 Darwin/16.7.0 botocore/1.5.35",
    "requestParameters": {
      "id": "****",
      "apiId": "a1b2c3d4e5f6g7h8i9jexample"
    },
    "responseElements": null,
    "requestID": "99999999-9999-9999-9999-999999999999",
  }
]
}

```



```

    "eventID": "99999999-9999-9999-9999-999999999999",
    "readOnly": false,
    "eventType": "AwsApiCall",
    "recipientAccountId": "111122223333"
  }
]
}

```

L'esempio seguente di voce di CloudTrail registro dimostra una mutazione GraphQL riuscita autorizzata con un autorizzatore di funzione Lambda personalizzato.

```

{
  "eventVersion": "1.10",
  "userIdentity": {
    "type": "Unknown"
  },
  "eventTime": "2024-11-06T15:42:30Z",
  "eventSource": "appsync.amazonaws.com",
  "eventName": "GraphQL",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "15.248.1.214",
  "userAgent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:131.0)
Gecko/20100101 Firefox/131.0",
  "requestParameters": null,
  "responseElements": null,
  "additionalEventData": {
    "operationName": "MyMutation",
    "authType": [
      "AWS_LAMBDA"
    ],
    "fieldAuthorizationResults": {
      "deniedFields": []
    }
  },
  "requestID": "c2d3768b-3446-40a1-bd95-8399fe776f96",
  "eventID": "21568be1-a1a8-4f43-b978-63cb4cc02a96",
  "readOnly": false,
  "resources": [
    {
      "accountId": "123456789012",
      "type": "AWS::AppSync::GraphQLApi",
      "ARN": "arn:aws:appsync:us-west-2:123456789012:apis/rxfqcxzi3nbvza2hsq4njqqq6u"
    }
  ]
}

```

```

],
"eventType": "AwsApiCall",
"managementEvent": false,
"recipientAccountId": "123456789012",
"eventCategory": "Data"
}

```

L'esempio seguente di voce di CloudTrail registro mostra un'operazione GraphQL parzialmente riuscita autorizzata con un autorizzatore di funzione Lambda personalizzato. Notate la `fieldAuthorizationResults.deniedFields` proprietà che specifica i campi negati.

```

{
  "eventVersion": "1.10",
  "userIdentity": {
    "type": "Unknown"
  },
  "eventTime": "2024-11-06T16:11:49Z",
  "eventSource": "appsync.amazonaws.com",
  "eventName": "GraphQL",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "15.248.1.214",
  "userAgent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:131.0) Gecko/20100101 Firefox/131.0",
  "requestParameters": null,
  "responseElements": null,
  "additionalEventData": {
    "operationName": "MyMutation",
    "authType": [
      "AWS_LAMBDA"
    ],
    "fieldAuthorizationResults": {
      "deniedFields": [
        "arn:aws:appsync:us-west-2:123456789012:apis/rxfqcxzi3nbvza2hsq4njqqq6u/types/Mutation/fields/createPost",
        "arn:aws:appsync:us-west-2:123456789012:apis/rxfqcxzi3nbvza2hsq4njqqq6u/types/Subscription/fields/onCreatePost",
        "arn:aws:appsync:us-west-2:123456789012:apis/rxfqcxzi3nbvza2hsq4njqqq6u/types/Post/fields/status"
      ]
    }
  },
  "requestID": "ae817c4c-66ba-4f64-92a5-ba9c9c341dcd",
  "eventID": "30109698-7605-476a-9dff-b7ed78d134dc",

```

```

"readOnly": false,
"resources": [
  {
    "accountId": "123456789012",
    "type": "AWS::AppSync::GraphQLApi",
    "ARN": "arn:aws:appsync:us-west-2:123456789012:apis/rxfqcxzi3nbvza2hsq4njqqq6u"
  }
],
"eventType": "AwsApiCall",
"managementEvent": false,
"recipientAccountId": "123456789012",
"eventCategory": "Data"
}

```

L'esempio seguente di voce di CloudTrail registro mostra un'operazione GraphQL non riuscita.

```

{
  "eventVersion": "1.10",
  "userIdentity": {
    "type": "Unknown"
  },
  "eventTime": "2024-11-06T15:51:11Z",
  "eventSource": "appsync.amazonaws.com",
  "eventName": "GraphQL",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "15.248.1.214",
  "userAgent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:131.0) Gecko/20100101 Firefox/131.0",
  "errorCode": "AccessDenied",
  "errorMessage": "{\n  \"errors\" : [ {\n    \"errorType\" : \"UnauthorizedException\", \n    \"message\" : \"You are not authorized to make this call.\" \n  } ]\n}",
  "requestParameters": null,
  "responseElements": null,
  "additionalEventData": {
    "operationName": "MyFullyDeniedLambdaMutation"
  },
  "requestID": "0bef3cf3-a48b-4de9-8b1f-038afb563516",
  "eventID": "b738651f-4ec0-4548-8fec-200c6b42842b",
  "readOnly": false,
  "resources": [
    {
      "accountId": "123456789012",
      "type": "AWS::AppSync::GraphQLApi",

```

```

    "ARN": "arn:aws:appsync:us-west-2:123456789012:apis/rxfqcxzi3nbvza2hsq4njqqq6u"
  }
],
"eventType": "AwsApiCall",
"managementEvent": false,
"recipientAccountId": "123456789012",
"eventCategory": "Data"
}

```

L'esempio seguente dimostra una richiesta GraphQL riuscita.

```

{
  "eventVersion": "1.10",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "AIDACKCEVSQ6C2EXAMPLE:jane_doe",
    "arn": "arn:aws:sts::123456789012:assumed-role/admin/jane_doe",
    "accountId": "123456789012",
    "sessionContext": {
      "sessionIssuer": {
        "type": "Role",
        "principalId": "AIDACKCEVSQ6C2EXAMPLE",
        "arn": "arn:aws:iam::123456789012:role/admin",
        "accountId": "123456789012",
        "userName": "jane_doe"
      },
      "attributes": {
        "creationDate": "2024-11-06T15:40:09Z",
        "mfaAuthenticated": "false"
      }
    }
  },
  "eventTime": "2024-11-06T16:03:43Z",
  "eventSource": "appsync.amazonaws.com",
  "eventName": "GraphQL",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "15.248.1.214",
  "userAgent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:131.0) Gecko/20100101 Firefox/131.0",
  "requestParameters": null,
  "responseElements": null,
  "additionalEventData": {
    "operationName": "IamFullSuccess",

```

```
{
  "authType": [
    "AWS_IAM"
  ],
  "fieldAuthorizationResults": {
    "allowedFields": [
      "arn:aws:appsync:us-west-2:123456789012:apis/rxfqcxzi3nbvza2hsq4njqqq6u/types/Mutation/fields/createSecondPostAllowed"
    ],
    "deniedFields": []
  }
},
"requestID": "edc6bbbf-6bf2-40f5-820f-ef444f12e0c1",
"eventID": "524656a5-0925-4370-9e7e-08888e9c299f",
"readOnly": false,
"resources": [
  {
    "accountId": "123456789012",
    "type": "AWS::AppSync::GraphQLApi",
    "ARN": "arn:aws:appsync:us-west-2:123456789012:apis/rxfqcxzi3nbvza2hsq4njqqq6u"
  }
],
"eventType": "AwsApiCall",
"managementEvent": false,
"recipientAccountId": "123456789012",
"eventCategory": "Data"
}
```

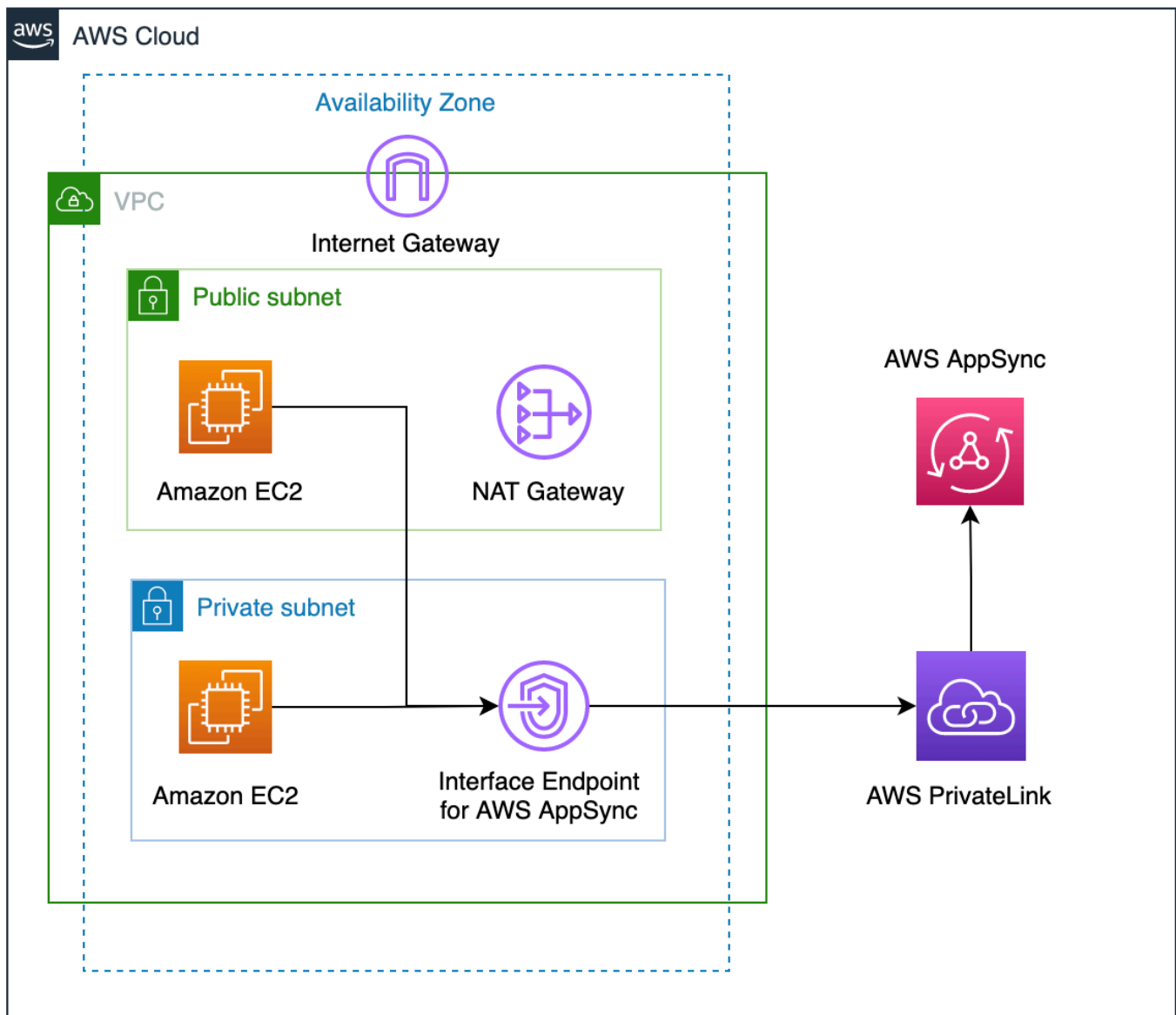
Utilizzo di AWS AppSync Private APIs

Se utilizzi Amazon Virtual Private Cloud (Amazon VPC), puoi creare AWS AppSync Private APIs, a APIs cui puoi accedere solo da un VPC. Con un'API privata, puoi limitare l'accesso dell'API alle tue applicazioni interne e connetterti agli endpoint GraphQL e Realtime senza esporre i dati pubblicamente.

Per stabilire una connessione privata tra il tuo VPC e il AWS AppSync servizio, devi creare endpoint VPC di interfaccia. Gli endpoint di interfaccia sono alimentati da [AWS PrivateLink](#), il che consente di accedere in modo privato AWS AppSync APIs senza un gateway Internet, un dispositivo NAT, una connessione VPN o una connessione. Direct Connect Le istanze del tuo VPC non necessitano di indirizzi IP pubblici con cui comunicare. AWS AppSync APIs Il traffico tra il tuo VPC e AWS AppSync non esce dalla AWS rete.

AWS AppSync supporta AWS PrivateLink le operazioni sia sul piano dati che sul piano di controllo:

- Data plane endpoint (`com.amazonaws.{region}.appsync-api`): fornisce accesso privato a GraphQL e APIs Real-time per interrogazioni, mutazioni e sottoscrizioni.
- Control plane endpoint (`com.amazonaws.{region}.appsync`): fornisce l'accesso privato alle operazioni di AWS AppSync gestione come la creazione APIs, l'aggiornamento di schemi e la configurazione delle fonti di dati.



Ci sono alcuni fattori aggiuntivi da considerare prima di abilitare le funzionalità dell'API privata:

- La configurazione degli endpoint dell'interfaccia VPC AWS AppSync con le funzionalità DNS private abilitate impedirà alle risorse del VPC di richiamare altri AWS AppSync utenti pubblici utilizzando l'URL API generato. APIs AWS AppSync Ciò è dovuto al fatto che la richiesta all'API pubblica viene instradata tramite l'endpoint dell'interfaccia, cosa che non è consentita per il pubblico. APIs Per richiamare public APIs in questo scenario, si consiglia di configurare nomi di dominio personalizzati su public APIs, che possono quindi essere utilizzati dalle risorse nel VPC per richiamare l'API pubblica.
- Il tuo AWS AppSync account privato APIs sarà disponibile solo dal tuo VPC. L'editor di query della AWS AppSync console sarà in grado di raggiungere l'API solo se la configurazione di rete del browser è in grado di indirizzare il traffico verso il VPC (ad esempio, connessione tramite VPN o superiore Direct Connect).
- Con un endpoint di interfaccia VPC per AWS AppSync, puoi accedere a qualsiasi API privata nello stesso AWS account e nella stessa regione. Per limitare ulteriormente l'accesso a Private APIs, puoi prendere in considerazione le seguenti opzioni:
 - Garantire che solo gli amministratori necessari possano creare interfacce endpoint VPC per AWS AppSync
 - Utilizzo di policy personalizzate per gli endpoint VPC per limitare quali elementi APIs possono essere richiamati dalle risorse del VPC.
 - Per le risorse nel VPC, ti consigliamo di utilizzare l'autorizzazione IAM per l'invocazione AWS AppSync APIs assicurandoti che alle risorse vengano assegnati ruoli limitati a APIs
- Quando si creano o si utilizzano policy che limitano i principi IAM, è necessario impostare il `authorizationType` metodo su `o. AWS_IAM NONE`

Creazione di contenuti privati AWS AppSync APIs

I seguenti passaggi mostrano come creare Private APIs nel AWS AppSync servizio.

Warning

È possibile abilitare le funzionalità dell'API privata solo durante la creazione dell'API. Questa impostazione non può essere modificata su un' AWS AppSync API o un'API AWS AppSync privata dopo la sua creazione.

1. Accedi a Console di gestione AWS e apri la [AppSync console](#).

- Nel pannello di controllo, scegliere Create API (Crea API).
2. Scegli Progetta un'API da zero, quindi scegli Avanti.
 3. Nella sezione API privata, scegli Usa le funzionalità dell'API privata.
 4. Configura il resto delle opzioni, esamina i dati dell'API, quindi scegli Crea.

Prima di poter utilizzare l'API AWS AppSync privata, devi configurare gli endpoint di interfaccia per AWS AppSync il tuo VPC. Tieni presente che sia l'API privata che il VPC devono trovarsi nello stesso AWS account e nella stessa regione.

Creazione di endpoint di interfaccia per AWS AppSync

Puoi creare endpoint di interfaccia per AWS AppSync utilizzare la console Amazon VPC o AWS Command Line Interface ().AWS CLI A seconda del caso d'uso, potrebbe essere necessario creare uno o entrambi i tipi di endpoint:

- Endpoint del piano dati: necessario per accedere a Private APIs dal tuo VPC
- Endpoint Control Plane: necessario per gestire AWS AppSync le risorse dal VPC utilizzando o AWS CLI SDKs

Per ulteriori informazioni, consultare [Creazione di un endpoint di interfaccia](#) nella Guida per l'utente di Amazon VPC.

Note

Assicurati di selezionare il servizio di endpoint VPC giusto; ce ne sono due per AppSync: `com.amazonaws.{region}.appsync-api` è quello richiesto per il servizio privato APIs mentre `com.amazonaws.{region}.appsync` viene utilizzato per la gestione delle API.

Console

1. Accedi Console di gestione AWS e apri la pagina [Endpoints](#) della console Amazon VPC.
2. Seleziona Crea endpoint.
 - a. Nel campo Categoria di servizio, verifica che AWS i servizi siano selezionati.
 - b. Nella tabella Servizi, scegli uno dei seguenti servizi:

- Per l'accesso al piano dati: `com.amazonaws.{region}.appsync-api`
- Per l'accesso al piano di controllo: `com.amazonaws.{region}.appsync`

Verificare che il valore della colonna `Type` sia `Interface`.

- Nel campo `VPC`, scegli un VPC e le relative sottoreti.
 - Per abilitare le funzionalità DNS private per l'endpoint dell'interfaccia, seleziona la casella di controllo `Abilita nome DNS`.
 - Nel campo `Gruppo di sicurezza`, scegli uno o più gruppi di sicurezza.
- Seleziona `Crea endpoint`.
 - Ripeti la procedura per creare il secondo tipo di endpoint, se necessario.

CLI

Utilizzare il comando [create-vpc-endpoint](#) e specificare l'ID VPC, il tipo di endpoint VPC (interfaccia), il nome del servizio, le sottoreti per utilizzare l'endpoint e i gruppi di sicurezza da associare alle interfacce di rete dell'endpoint.

Crea un endpoint del piano dati:

```
$ aws ec2 create-vpc-endpoint --vpc-id vpc-ec43eb89 \
--vpc-endpoint-type Interface \
--service-name com.amazonaws.{region}.appsync-api \
--subnet-id subnet-abababab --security-group-id sg-1a2b3c4d
```

Crea un endpoint del piano di controllo:

```
$ aws ec2 create-vpc-endpoint --vpc-id vpc-ec43eb89 \
--vpc-endpoint-type Interface \
--service-name com.amazonaws.{region}.appsync \
--subnet-id subnet-abababab --security-group-id sg-1a2b3c4d
```

Per utilizzare l'opzione DNS privato, devi impostare `enableDnsSupportattributes` i valori `enableDnsHostnames` and del tuo VPC. Per ulteriori informazioni, consulta [Visualizzazione e aggiornamento del supporto DNS per il VPC](#) nella Guida per l'utente di Amazon VPC. Se abiliti le funzionalità DNS private per l'endpoint di interfaccia, puoi effettuare richieste all'endpoint API AWS

AppSync GraphQL e Real-time utilizzando gli endpoint DNS pubblici predefiniti utilizzando il formato seguente:

```
https://{api_url_identifier}.appsync-api.{region}.amazonaws.com/graphql
```

Per le operazioni del piano di controllo, puoi utilizzare l'endpoint di servizio standard: AWS AppSync

```
https://appsync.{region}.amazonaws.com
```

Per ulteriori informazioni sugli endpoint di servizio, vedere [Service endpoints and quotas](#) nel General Reference.AWS

Per ulteriori informazioni sulle interazioni dei servizi con gli endpoint di interfaccia, consulta [Accedere a un servizio tramite un endpoint di interfaccia](#) nella Amazon VPC User Guide.

Per informazioni sulla creazione e configurazione di un endpoint utilizzando AWS CloudFormation, consulta la risorsa [AWS: :EC2::](#) nella Guida per l'utente. VPC Endpoint AWS CloudFormation

Esempi avanzati di

Se abiliti le funzionalità DNS private per l'endpoint di interfaccia, puoi effettuare richieste all'endpoint API AWS AppSync GraphQL e Real-time utilizzando gli endpoint DNS pubblici predefiniti utilizzando il formato seguente:

```
https://{api_url_identifier}.appsync-api.{region}.amazonaws.com/graphql
```

Utilizzando i nomi di host DNS pubblici degli endpoint VPC dell'interfaccia, l'URL di base per richiamare l'API sarà nel seguente formato:

```
https://{vpc_endpoint_id}-{endpoint_dns_identifier}.appsync-api.  
{region}.vpce.amazonaws.com/graphql
```

Puoi anche utilizzare il nome host DNS specifico per AZ se hai distribuito un endpoint nell'AZ:

```
https://{vpc_endpoint_id}-{endpoint_dns_identifier}-{az_id}.appsync-api.  
{region}.vpce.amazonaws.com/graphql.
```

L'utilizzo del nome DNS pubblico dell'endpoint VPC richiederà il passaggio del nome host dell'endpoint AWS AppSync API come Host o come intestazione alla richiesta. x-appsync-

domain [Questi esempi utilizzano uno schema creato nella TodoAPI guida Launch a sample schema:](#)

```
curl https://{vpc_endpoint_id}-{endpoint_dns_identifier}.appsync-api.
{region}.vpce.amazonaws.com/graphql \
-H "Content-Type:application/graphql" \
-H "x-api-key:da2-{xxxxxxxxxxxxxxxxxxxxxxxxxxxx}" \
-H "Host:{api_url_identifier}.appsync-api.{region}.amazonaws.com" \
-d '{"query":"mutation add($createtodoinput: CreateTodoInput!) {\n createTodo(input:
$createtodoinput) {\n id\n name\n where\n when\n description\n }\n}", "variables":
{"createtodoinput":{"name":"My first GraphQL task", "when":"Friday Night", "where":"Day
1", "description":"Learn more about GraphQL"}}}'
```

Negli esempi seguenti, utilizzeremo l'app Todo generata nella guida [allo schema di esempio Launch a](#). Per testare l'API Todo di esempio, utilizzeremo il DNS privato per richiamare l'API. [Puoi usare qualsiasi strumento da riga di comando a tua scelta; questo esempio usa curl per inviare domande e mutazioni e wscat per configurare gli abbonamenti.](#) Per emulare il nostro esempio, sostituisci i valori tra parentesi nei comandi seguenti con i valori { } corrispondenti del tuo account. AWS

Test dell'operazione di mutazione — Richiesta **createTodo**

```
curl https://{api_url_identifier}.appsync-api.{region}.amazonaws.com/graphql \
-H "Content-Type:application/graphql" \
-H "x-api-key:da2-{xxxxxxxxxxxxxxxxxxxxxxxxxxxx}" \
-d '{"query":"mutation add($createtodoinput: CreateTodoInput!) {\n createTodo(input:
$createtodoinput) {\n id\n name\n where\n when\n description\n }\n}", "variables":
{"createtodoinput":{"name":"My first GraphQL task", "when":"Friday Night", "where":"Day
1", "description":"Learn more about GraphQL"}}}'
```

Test dell'operazione di mutazione — Risposta **createTodo**

```
{
  "data": {
    "createTodo": {
      "id": "<todo-id>",
      "name": "My first GraphQL task",
      "where": "Day 1",
      "when": "Friday Night",
      "description": "Learn more about GraphQL"
    }
  }
}
```

```
}
```

Operazione di interrogazione di test — **listTodos** Richiesta

```
curl https://{api_url_identifrier}.appsync-api.{region}.amazonaws.com/graphql \
-H "Content-Type:application/graphql" \
-H "x-api-key:da2-{xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}" \
-d '{"query":"query ListTodos {\n listTodos {\n items {\n description\n id\n name\n when\n where\n }\n }\n\n","variables":{"createtodoinput":{"name":"My first\n GraphQL task","when":"Friday Night","where":"Day 1","description":"Learn more about\n GraphQL"}}}'
```

Operazione di interrogazione di test — **listTodos** Richiesta

```
{
  "data": {
    "listTodos": {
      "items": [
        {
          "description": "Learn more about GraphQL",
          "id": "<todo-id>",
          "name": "My first GraphQL task",
          "when": "Friday night",
          "where": "Day 1"
        }
      ]
    }
  }
}
```

Test del funzionamento dell'abbonamento — Iscrizione alla **createTodo** mutazione

Per configurare gli abbonamenti GraphQL in AWS AppSync, consulta [Creazione di un client in tempo reale. WebSocket](#) [Da un'istanza Amazon EC2 in un VPC, puoi testare il tuo endpoint di abbonamento a AWS AppSync Private API utilizzando wscat](#). L'esempio seguente utilizza un per l'autorizzazione.

API KEY

```
$ header=`echo '{"host":"{api_url_identifrier}.appsync-api.{region}.amazonaws.com","x-api-key":"da2-{xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}"' | base64 | tr -d '\n'`
$ wscat -p 13 -s graphql-ws -c "wss://{api_url_identifrier}.appsync-realtime-api.us-west-2.amazonaws.com/graphql?header=$header&payload=e30="
Connected (press CTRL+C to quit)
```

```
> {"type": "connection_init"}
< {"type": "connection_ack", "payload": {"connectionTimeoutMs": 300000}}
< {"type": "ka"}
> {"id": "f7a49717", "payload": {"data": {"\query\": "\subscription
onCreateTodo {onCreateTodo {description id name where when}}\",
\variables\": {}}, "extensions": {"authorization": {"x-api-key": "da2-
{xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}", "host": "{api_url_identifier}.appsync-api.
{region}.amazonaws.com"}}}, "type": "start"}
< {"id": "f7a49717", "type": "start_ack"}
```

In alternativa, usa il nome di dominio dell'endpoint VPC assicurandoti di specificare l'intestazione Host nel wscat comando per stabilire il websocket:

```
$ header=`echo '{"host": "{api_url_identifier}.appsync-api.{region}.amazonaws.com", "x-
api-key": "da2-{xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}"' | base64 | tr -d '\n'`
$ wscat -p 13 -s graphql-ws -c "wss://{vpc_endpoint_id}-
{endpoint_dns_identifier}.appsync-api.{region}.vpce.amazonaws.com/graphql?header=
$header&payload=e30=" --header Host:{api_url_identifier}.appsync-realtime-api.us-
west-2.amazonaws.com
Connected (press CTRL+C to quit)
> {"type": "connection_init"}
< {"type": "connection_ack", "payload": {"connectionTimeoutMs": 300000}}
< {"type": "ka"}
> {"id": "f7a49717", "payload": {"data": {"\query\": "\subscription
onCreateTodo {onCreateTodo {description id priority title}}\",
\variables\": {}}, "extensions": {"authorization": {"x-api-key": "da2-
{xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}", "host": "{api_url_identifier}.appsync-api.
{region}.amazonaws.com"}}}, "type": "start"}
< {"id": "f7a49717", "type": "start_ack"}
```

Esegui il codice di mutazione seguente:

```
curl https://{api_url_identifier}.appsync-api.{region}.amazonaws.com/graphql \
-H "Content-Type:application/graphql" \
-H "x-api-key:da2-{xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}" \
-d '{"query": "mutation add($createtodoinput: CreateTodoInput!) {\n createTodo(input:
$createtodoinput) {\n id\n name\n where\n when\n description\n }\n}", "variables":
{"createtodoinput": {"name": "My first GraphQL task", "when": "Friday Night", "where": "Day
1", "description": "Learn more about GraphQL"}}}'
```

Successivamente, viene attivato un abbonamento e la notifica del messaggio viene visualizzata come mostrato di seguito:

```
< {"id":"f7a49717","type":"data","payload":{"data":{"onCreateTodo":{"description":"Go to the shops","id":"169ce516-b7e8-4a6a-88c1-ab840184359f","priority":5,"title":"Go to the shops"}}}}}
```

Esempi di piani di controllo

Con l'endpoint VPC del piano di controllo configurato, puoi gestire AWS AppSync le risorse dall'interno del tuo VPC utilizzando o. AWS CLI SDKs Ecco alcuni esempi di operazioni comuni sul piano di controllo:

Creazione di un'API utilizzando AWS CLI

```
aws appsync create-graphql-api \  
  --name "MyPrivateAPI" \  
  --authentication-type API_KEY \  
  --visibility PRIVATE
```

Aggiornamento di uno schema

```
aws appsync start-schema-creation \  
  --api-id {api-id} \  
  --definition file://schema.graphql
```

Creazione di una fonte di dati

```
aws appsync create-data-source \  
  --api-id {api-id} \  
  --name "MyDataSource" \  
  --type AWS_LAMBDA \  
  --lambda-config lambdaFunctionArn=arn:aws:lambda:{region}:  
{account}:function:MyFunction
```

Quando si utilizza l'endpoint del piano di controllo con DNS privato abilitato, questi comandi verranno instradati automaticamente attraverso l'endpoint VPC. Se il DNS privato non è abilitato, puoi specificare l'URL dell'endpoint:

```
aws appsync create-graphql-api \  
  --endpoint-url https://{vpc_endpoint_id}-{endpoint_dns_identifier}.appsync.  
{region}.vpce.amazonaws.com \  
  --visibility PRIVATE
```

```
--name "MyPrivateAPI" \  
--authentication-type API_KEY \  
--visibility PRIVATE
```

Utilizzo delle policy IAM per limitare la creazione di API pubbliche

AWS AppSync supporta [Conditionle istruzioni](#) IAM da utilizzare con Private APIs.

Il `visibility` campo può essere incluso nelle dichiarazioni politiche di IAM per l'operazione `appsync:CreateGraphqlApi` volta a controllare quali ruoli e utenti IAM possono creare elementi privati e pubblici APIs. Ciò offre a un amministratore IAM la possibilità di definire una policy IAM che consenta solo a un utente di creare un'API GraphQL privata. Un utente che tenta di creare un'API pubblica riceverà un messaggio non autorizzato.

Ad esempio, un amministratore IAM potrebbe creare la seguente dichiarazione di policy IAM per consentire la creazione di Private: APIs

```
{  
  "Sid": "AllowPrivateAppSyncApis",  
  "Effect": "Allow",  
  "Action": "appsync:CreateGraphqlApi",  
  "Resource": "*",  
  "Condition": {  
    "ForAnyValue:StringEquals": {  
      "appsync:Visibility": "PRIVATE"  
    }  
  }  
}
```

Un amministratore IAM può anche aggiungere la seguente [politica di controllo del servizio](#) per impedire a tutti gli utenti di un' AWS organizzazione di creare qualcosa di AWS AppSync APIs diverso da Private APIs:

```
{  
  "Sid": "BlockNonPrivateAppSyncApis",  
  "Effect": "Deny",  
  "Action": "appsync:CreateGraphqlApi",  
  "Resource": "*",  
  "Condition": {  
    "ForAnyValue:StringNotEquals": {  
      "appsync:Visibility": "PRIVATE"  
    }  
  }  
}
```

```
}  
  }  
}
```

Supporto VPC PrivateLink

Il supporto per VPC PrivateLink è disponibile in. AWS AppSync PrivateLink consente di utilizzare e interagire con un AWS Servizio senza che il traffico esca dalla rete. AWS

AWS AppSync supporta AWS PrivateLink le operazioni sia sul piano dati che sul piano di controllo.

- VPCE endpoint (`appsync.<region>.vpce.amazonaws.com`): fornisce l'accesso VPC alle operazioni del piano dati e del piano di controllo come segue:
 - `appsync` per le operazioni sul piano di controllo
 - `appsync-api` per le operazioni sul piano dati

Condivisione di AWS AppSync GraphQL APIs

AWS AppSync si integra con AWS Resource Access Manager (AWS RAM) per consentire la condivisione delle risorse. AWS RAM è un servizio che consente di condividere azioni di invoke (operazioni di interrogazione, mutazione e sottoscrizione e richieste di connessione all' WebSocket endpoint in tempo reale) su GraphQL AWS AppSync con altri o tramite. APIs Account AWS AWS Organizations Con AWS RAM, condividi le risorse di tua proprietà creando una condivisione di risorse. Una condivisione delle risorse specifica le risorse da condividere e gli utenti con cui condividerle. I consumatori possono includere quanto segue.

- Specifico Account AWS all'interno o all'esterno della sua organizzazione in AWS Organizations
- Un'unità organizzativa all'interno della propria organizzazione in AWS Organizations
- Un'intera organizzazione in AWS Organizations

Per ulteriori informazioni in merito AWS RAM, consulta la [Guida AWS Resource Access Manager per l'utente](#).

Argomenti

- [Prerequisiti per la condivisione AWS AppSync di GraphQL APIs](#)
- [Condividi AWS AppSync GraphQL APIs](#)

- [Smetti di condividere AWS AppSync GraphQL APIs](#)
- [Eventi multi-account](#)

Prerequisiti per la condivisione AWS AppSync di GraphQL APIs

La condivisione di AWS AppSync GraphQL APIs presenta i seguenti prerequisiti.

- Per condividere un'API AWS AppSync GraphQL, devi possederla nel tuo Account AWS. Ciò significa che l'API AWS AppSync GraphQL deve essere allocata o fornita nel tuo account.
- Per condividere un'API AWS AppSync GraphQL con la tua organizzazione o un'unità organizzativa in AWS Organizations, devi abilitare la condivisione con AWS Organizations. Per ulteriori informazioni, consulta [Abilitare la condivisione delle risorse con AWS Organizations](#) nella Guida per l'utente di AWS Resource Access Manager.

Condividi AWS AppSync GraphQL APIs

Per condividere un'API AWS AppSync GraphQL, inizia creando una condivisione di risorse utilizzando AWS Resource Access Manager. Una condivisione di risorse specifica le risorse da condividere, i consumatori con cui vengono condivise e quali azioni possono eseguire i responsabili. Quando condividi un'API AWS AppSync GraphQL di tua proprietà con altri Account AWS, consenti a quegli account di chiamare quell'AWS AppSync API nel tuo Account AWS.

Se fai parte di un'organizzazione e la condivisione all'interno dell'organizzazione è abilitata, ai consumatori dell'organizzazione viene automaticamente concesso l'accesso alla risorsa condivisa. AWS Organizations. In caso contrario, i consumatori ricevono un invito a partecipare alla condivisione di risorse e ottengono l'accesso alla risorsa condivisa dopo aver accettato l'invito.

Considerazioni sulla condivisione

- Puoi condividere solo AWS AppSync GraphQL APIs, non altri tipi di API come Event. APIs
- Puoi condividere solo AWS AppSync GraphQL APIs che ha AWS_IAM una delle modalità di autorizzazione configurate sull'API.

Se AWS_IAM viene rimossa dall'elenco delle modalità di autorizzazione per un'API AppSync GraphQL condivisa, anche se la condivisione di risorse potrebbe ancora esistere, verrà resa inefficace.

- Puoi condividere AWS AppSync APIs GraphQL sia pubblico che privato.

- APIs È sempre possibile accedere a Private AWS AppSync GraphQL tramite endpoint VPC VPCs nell'origine e sono supportate tutte le modalità di autorizzazione Account AWS, non solo. AWS_IAM
- Per AWS AppSync GraphQL condiviso APIs, le autorizzazioni sono gestite solo per la risorsa API e non supportano autorizzazioni granulari per campo, tipo e risorse sul campo. Quando condividi un'API, condividi l'ARN dell'API e ARNs per tutti i suoi tipi e campi.

Crea una condivisione di risorse di tua proprietà utilizzando la console AWS RAM

Per condividere un'API AWS AppSync GraphQL, utilizzare la procedura descritta in [Creazione di una condivisione di risorse](#) nella Guida per l'AWS Resource Access Manager utente, utilizzando il nome dell'RAMautorizzazione. AWSRAMPermissionAppSyncGraphQLApiInvokeAccess

Crea e utilizza un'autorizzazione gestita dal cliente per condividere un'API AWS AppSync GraphQL privata utilizzando la console AWS RAM

Per condividere un'API AWS AppSync GraphQL privata, crea un'autorizzazione gestita dal cliente utilizzando la procedura descritta in [Creazione e utilizzo delle autorizzazioni gestite dal cliente](#) nella Guida per l'AWS Resource Access Manager utente.

Ad esempio, il proprietario dell'Account A desidera concedere ai principali dell'Account B l'autorizzazione ad accedere a un'API AWS AppSync GraphQL privata PrivateApi (A) per le chiamate effettuate tramite VPCE-B (un endpoint VPC di proprietà dell'Account B). In questo caso, il proprietario dell'Account A deve creare un'autorizzazione gestita dal cliente come segue. AWS RAM

```
{
  "Effect": "Allow",
  "Action": ["appsync:GraphQL"],
  "Condition": {
    "StringEqualsIgnoreCase": {
      "aws:SourceVpce": [
        "VPCE-B"
      ]
    }
  }
}
```

Supponiamo che questa nuova AWS RAM autorizzazione gestita dal cliente sia denominata `private-api-A-access-via-vpce-b`.

Per abilitare l'accesso a PrivateApiA via su più accountVPCE-B, il cliente può creare una condivisione di AWS RAM risorse con i seguenti parametri e l'autorizzazione gestita dal cliente nell'esempio precedente.

- Tipo di risorsa: `appsync:Apis`
- Risorsa: `arn:aws:appsync:us-west-2:A:apis/PrivateApiA`
- Autorizzazione: `private-api-A-access-via-vpce-b` (autorizzazione gestita dal cliente)
- Preside: Account: B

Crea una condivisione di risorse di tua proprietà utilizzando il AWS CLI

Per condividere un'API AWS AppSync GraphQL utilizzando il AWS CLI, usa il `create-resource-share` comando with `arn:aws:ram::aws:permission/AWSRAMPermissionAppSyncApiInvokeAccess` come valore per lo `--permission-arns` switch.

Per un elenco completo dei comandi disponibili per AWS RAM, consulta il riferimento [AWS RAM CLI](#).

Smetti di condividere AWS AppSync GraphQL APIs

Per interrompere la condivisione di AWS AppSync GraphQL di APIs cui sei proprietario, devi eliminare la condivisione di risorse o aggiornare i principali con cui hai condiviso la risorsa. Fate riferimento alla documentazione nelle sezioni seguenti per l'azione che desiderate eseguire.

Per interrompere la condivisione di una risorsa di tua proprietà utilizzando la AWS RAM console

Consulta [Update a resource share](#) nella Guida per l'utente di AWS Resource Access Manager .

Per interrompere la condivisione di una risorsa di tua proprietà, utilizza il AWS CLI

Utilizza il comando [disassociate-resource-share](#).

Per eliminare una condivisione di risorse di tua proprietà utilizzando la AWS RAM console

Vedi [Eliminazione di una condivisione di risorse](#) nella Guida per l'AWS Resource Access Manager utente.

Per eliminare una condivisione di risorse di tua proprietà, utilizza il AWS CLI

Utilizza il comando [delete-resource-share](#).

Per un elenco completo dei comandi disponibili per AWS RAM, consulta il riferimento [AWS RAM CLI](#).

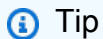
Eventi multi-account

Puoi attivare la registrazione degli eventi AWS CloudTrail dei dati per il monitoraggio e il controllo dell'attività dell'API AWS AppSync GraphQL tra account. DataPlane Per ulteriori informazioni, consultare [Registrazione di eventi di dati](#) nella Guida per l'utente di AWS CloudTrail .

Configurazione della complessità dell'esecuzione, della profondità delle query e dell'introspezione di GraphQL con AWS AppSync

AWS AppSync consente di abilitare o disabilitare le funzionalità di introspezione e di impostare limiti alla quantità di livelli e resolver annidati in una singola query.

Utilizzo della funzione di introspezione



Tip

[Per ulteriori informazioni sull'introspezione in GraphQL, consulta questo articolo sul sito Web della fondazione GraphQL.](#)

Per impostazione predefinita, GraphQL consente di utilizzare l'introspezione per interrogare lo schema stesso per scoprirne i tipi, i campi, le query, le mutazioni, le sottoscrizioni, ecc. Questa è una funzionalità importante per imparare come i dati vengono modellati ed elaborati dal servizio GraphQL. Tuttavia, ci sono alcuni aspetti da considerare quando si ha a che fare con l'introspezione. Potresti avere un caso d'uso che trarrebbe vantaggio dalla disattivazione dell'introspezione, ad esempio un caso in cui i nomi dei campi potrebbero essere riservati o nascosti o lo schema API completo è destinato a non essere documentato per i consumatori. In questi casi, la pubblicazione dei dati dello schema tramite l'introspezione potrebbe causare la fuga di dati intenzionalmente privati.

Per evitare che ciò accada, puoi disabilitare l'introspezione. Ciò impedirà a parti non autorizzate di utilizzare campi di introspezione nello schema. Tuttavia, è importante notare che l'introspezione è utile ai team di sviluppo per apprendere come vengono elaborati i dati del loro servizio. Internamente, potrebbe essere utile mantenere attiva l'introspezione e disabilitarla nel codice di produzione come ulteriore livello di sicurezza. Un altro modo per gestire questo problema è aggiungere un metodo di autorizzazione, che prevede anche. AWS AppSync Per ulteriori informazioni, consulta [Autorizzazione](#).

AWS AppSync consente di abilitare o disabilitare l'introspezione a livello di API. Per abilitare o disabilitare l'introspezione, procedi come segue:

1. [Accedi a Console di gestione AWS e apri la AppSync console.](#)
2. Nella APIspagina, scegli il nome di un'API GraphQL.
3. Nella home page dell'API, nel riquadro di navigazione, scegli Impostazioni.
4. Nelle configurazioni API, scegli Modifica.
5. In Query di introspezione, procedi come segue:
 - Attiva o disattiva Abilita le interrogazioni di introspezione.
6. Scegli Save (Salva).

Quando l'introspezione è abilitata (il comportamento predefinito), l'utilizzo del sistema di introspezione funzionerà normalmente. Ad esempio, l'immagine seguente mostra un `__schema` campo che elabora tutti i tipi disponibili nello schema:



Quando si disabilita questa funzione, nella risposta verrà invece visualizzato un errore di convalida:



Configurazione dei limiti di profondità delle interrogazioni

In alcuni momenti potresti aver bisogno di un controllo più granulare sul funzionamento dell'API durante un'operazione. Uno di questi controlli consiste nell'aggiungere un limite alla quantità di livelli annidati che una query può elaborare. Per impostazione predefinita, le interrogazioni sono in grado di elaborare un numero illimitato di livelli annidati. Limitare le interrogazioni a un determinato numero di livelli annidati ha potenziali implicazioni per le prestazioni e la flessibilità del progetto. Eseguire la seguente query:

```

query MyQuery {
  L1: nextLayer {
    L2: nextLayer {
      L3: nextLayer {
        L4: value
      }
    }
  }
}

```

Il progetto potrebbe richiedere la limitazione delle interrogazioni a o per qualche scopo. L1 L2 Per impostazione predefinita, l'intera query da L1 a L4 verrebbe elaborata senza alcun modo per controllarla. Impostando un limite, è possibile impedire alle query di accedere a qualsiasi elemento oltre il livello specificato.

Per aggiungere un limite di profondità delle query, procedi come segue:

1. Accedi a Console di gestione AWS e apri la [AppSynconconsole](#).
2. Nella APIspagina, scegli il nome di un'API GraphQL.
3. Nella home page dell'API, nel riquadro di navigazione, scegli Impostazioni.
4. Nelle configurazioni API, scegli Modifica.
5. In Profondità della query, procedi come segue:
 - a. Attiva o disattiva Abilita la profondità delle interrogazioni.
 - b. In Profondità massima, imposta il limite di profondità. Questo valore può essere compreso tra 1 e 75.
6. Scegli Save (Salva).

Quando viene impostato un limite, il superamento del limite superiore genererà un `QueryDepthLimitReached` errore. Ad esempio, l'immagine seguente mostra una query con un limite di profondità 2 che consente di superare il limite fino al terzo (L3) e al quarto (L4) livello:



Tieni presente che i campi possono ancora essere contrassegnati come annullabili o non annullabili nello schema. Se un campo non annullabile riceve un `QueryDepthLimitReached` errore, tale errore verrà generato nel primo campo padre annullabile.

Configurazione dei limiti di conteggio dei resolver

È inoltre possibile controllare il numero di resolver che ciascuna query può elaborare. Analogamente alla profondità della query, puoi impostare un limite a questo importo. Prendi la seguente query che contiene tre resolver:

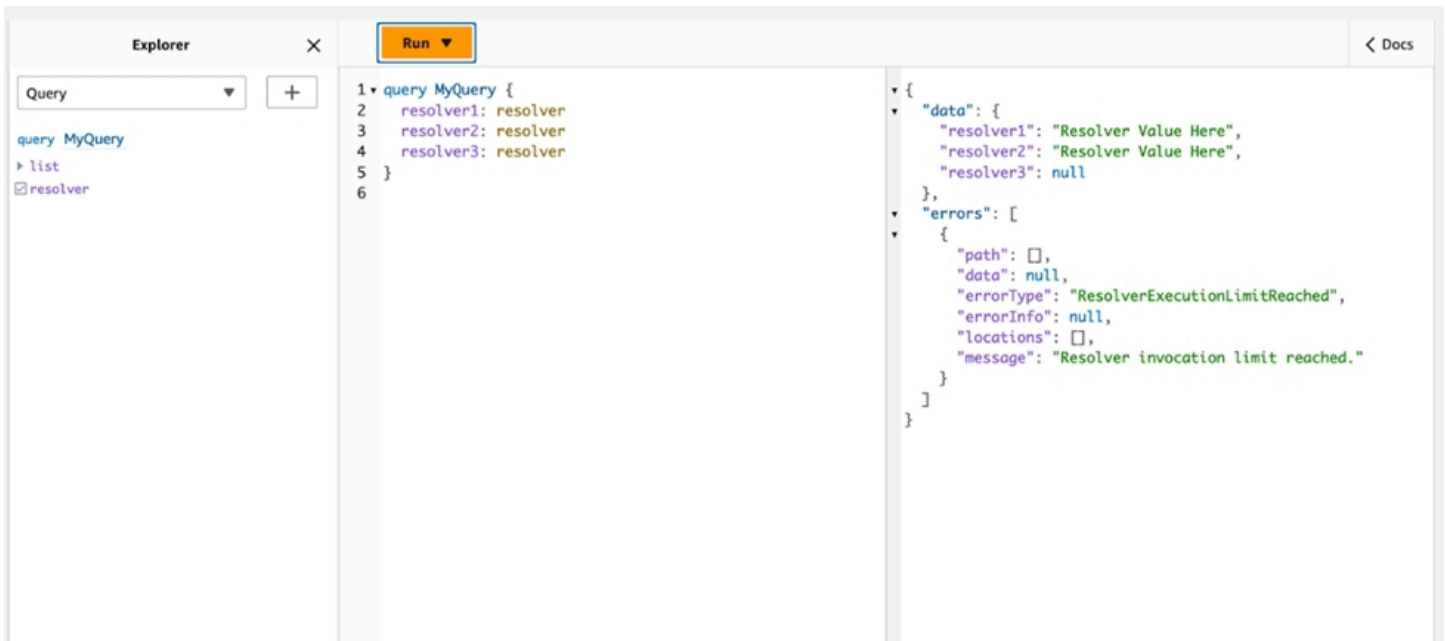
```
query MyQuery {  
  resolver1: resolver  
  resolver2: resolver  
  resolver3: resolver  
}
```

Per impostazione predefinita, ogni query può elaborare fino a 10000 resolver. Nell'esempio precedente, `resolver1`, `resolver2`, e `resolver3` verranno elaborati. Tuttavia, il progetto potrebbe richiedere di limitare ogni query alla gestione di uno o due resolver in totale. Impostando un limite, potete dire alla query di non gestire alcun resolver oltre un certo numero, come i resolver del primo (`resolver1`) o del secondo (`resolver2`).

Per aggiungere un limite di numero di resolver, procedi come segue:

1. [Accedi a Console di gestione AWS e apri la AppSync console.](#)
2. Nella API pagina, scegli il nome di un'API GraphQL.
3. Nella home page dell'API, nel riquadro di navigazione, scegli Impostazioni.
4. Nelle configurazioni API, scegli Modifica.
5. Sotto il limite di numero di Resolver, procedi come segue:
 - a. Attiva Abilita il conteggio dei resolver.
 - b. In Numero massimo di resolver, imposta il limite di conteggio. Questo valore può essere compreso tra e1. 10000
6. Scegli Save (Salva).

Analogamente al limite di profondità della query, il superamento del limite di resolver configurato causa la fine della query con un `ResolverExecutionLimitReached` errore sui resolver aggiuntivi. Nell'immagine seguente, una query con un limite di numero di resolver pari a 2 tenta di elaborare tre resolver. A causa del limite, il terzo resolver genera un errore e non viene eseguito.



Utilizzo delle variabili di ambiente in AWS AppSync

Puoi utilizzare le variabili di ambiente per regolare il comportamento dei AWS AppSync resolver e delle funzioni senza aggiornare il codice. Le variabili di ambiente sono coppie di stringhe memorizzate nella configurazione dell'API che vengono rese disponibili ai resolver e alle funzioni per utilizzarle in fase di esecuzione. Sono particolarmente utili nelle situazioni in cui è necessario fare riferimento a dati di configurazione disponibili solo durante la configurazione iniziale ma che devono essere utilizzati dai resolver e dalle funzioni durante l'esecuzione. Le variabili di ambiente espongono i dati di configurazione contenuti nel codice, riducendo in tal modo la necessità di codificare tali valori.

Note

Per aumentare la sicurezza del database, si consiglia di utilizzare [Secrets Manager](#) o [AWS Systems Manager Parameter Store](#) anziché le variabili di ambiente per archiviare credenziali o informazioni riservate. Per sfruttare questa funzionalità, consulta [Richiamo di AWS servizi con fonti di dati AWS AppSync HTTP](#).

Le variabili di ambiente devono seguire diversi comportamenti e regole per funzionare correttamente:

- Sia i JavaScript resolver/funzioni che i modelli VTL supportano le variabili di ambiente.
- Le variabili di ambiente non vengono valutate prima dell'invocazione della funzione.

- Le variabili di ambiente supportano solo valori di stringa.
- Qualsiasi valore definito in una variabile di ambiente viene considerato una stringa letterale e non espanso.
- Le valutazioni delle variabili dovrebbero idealmente essere eseguite nel codice della funzione.

Configurazione delle variabili di ambiente (console)

Puoi configurare le variabili di ambiente per la tua API AWS AppSync GraphQL creando la variabile e definendo la sua coppia chiave-valore. I resolver e le funzioni utilizzeranno il nome chiave della variabile di ambiente per recuperare il valore in fase di esecuzione. Per impostare le variabili di ambiente nella console: AWS AppSync

1. Accedi a Console di gestione AWS e apri la [AppSyncconsole](#).
2. Nella APIspagina, scegli il nome di un'API GraphQL.
3. Nella home page dell'API, nel riquadro di navigazione, scegli Impostazioni.
4. In Variabili di ambiente, scegli Aggiungi variabile di ambiente.
5. Scegli Add environment variable (Aggiungi variabile d'ambiente).
6. Inserisci una coppia chiave valore.
7. Se necessario, ripeti i passaggi 5 e 6 per aggiungere altri valori chiave. Se devi rimuovere un valore chiave, scegli l'opzione Rimuovi e le chiavi da rimuovere.
8. Scegli Invia.

Tip

Ci sono alcune regole da seguire quando si creano chiavi e valori:

- Le chiavi devono iniziare con una lettera.
- Le chiavi devono contenere almeno due caratteri.
- Le chiavi possono contenere solo lettere, numeri e il carattere di sottolineatura (_).
- I valori possono avere una lunghezza massima di 512 caratteri.
- Puoi configurare fino a 50 coppie chiave-valore in un'API GraphQL.

Configurazione delle variabili di ambiente (API)

Per impostare una variabile di ambiente utilizzando APIs, puoi usare `PutGraphqlApiEnvironmentVariables`. Il comando CLI corrispondente è `put-graphql-api-environment-variables`.

Per recuperare una variabile di ambiente utilizzando APIs, puoi usare `GetGraphqlApiEnvironmentVariables`. Il comando CLI corrispondente è `get-graphql-api-environment-variables`.

Il comando deve contenere l'ID API e l'elenco delle variabili di ambiente:

```
aws appsync put-graphql-api-environment-variables \
  --api-id "<api-id>" \
  --environment-variables '{"key1":"value1","key2":"value2", ...}'
```

L'esempio seguente imposta due variabili di ambiente in un'API con l'ID di `abcdefghijklmnopqrstuvwxyz` utilizzando il `put-graphql-api-environment-variables` comando:

```
aws appsync put-graphql-api-environment-variables \
  --api-id "abcdefghijklmnopqrstuvwxyz" \
  --environment-variables '{"USER_TABLE":"users_prod","DEBUG":"true"}'
```

Si noti che quando si applicano variabili di ambiente con il `put-graphql-api-environment-variables` comando, il contenuto della struttura delle variabili di ambiente viene sovrascritto; ciò significa che le variabili di ambiente esistenti andranno perse. Per mantenere le variabili di ambiente esistenti quando ne aggiungi di nuove, includi tutte le coppie chiave-valore esistenti insieme a quelle nuove nella richiesta. Utilizzando l'esempio precedente, se desideri aggiungere `"EMPTY": ""`, puoi fare quanto segue:

```
aws appsync put-graphql-api-environment-variables \
  --api-id "abcdefghijklmnopqrstuvwxyz" \
  --environment-variables '{"USER_TABLE":"users_prod","DEBUG":"true", "EMPTY":""}'
```

Per recuperare la configurazione corrente, usa il `get-graphql-api-environment-variables` comando:

```
aws appsync get-graphql-api-environment-variables --api-id "<api-id>"
```

Utilizzando l'esempio precedente, è possibile utilizzare il seguente comando:

```
aws appsync get-graphql-api-environment-variables --api-id "abcdefghijklmnoprstuvwxyz"
```

Il risultato mostrerà l'elenco delle variabili di ambiente insieme ai loro valori chiave:

```
{
  "environmentVariables": {
    "USER_TABLE": "users_prod",
    "DEBUG": "true",
    "EMPTY": ""
  }
}
```

Configurazione delle variabili di ambiente (CFN)

È possibile utilizzare il modello seguente per creare variabili di ambiente:

```
AWSTemplateFormatVersion: 2010-09-09
Resources:
  GraphQLApiWithEnvVariables:
    Type: "AWS::AppSync::GraphQLApi"
    Properties:
      Name: "MyApiWithEnvVars"
      AuthenticationType: "AWS_IAM"
      EnvironmentVariables:
        EnvKey1: "non-empty"
        EnvKey2: ""
```

variabili di ambiente e unite APIs

Le variabili di ambiente definite in Source APIs sono disponibili anche in APIs Merged. Le variabili di ambiente in Merged APIs sono di sola lettura e non possono essere aggiornate. Tieni presente che le chiavi delle variabili di ambiente devono essere univoche in tutto Source APIs affinché le unioni abbiano successo; le chiavi duplicate comporteranno sempre un errore di unione.

Recupero di variabili di ambiente

Per recuperare le variabili di ambiente nel codice della funzione, recuperate il valore dall'`ctx.env` oggetto nei resolver e nelle funzioni. Di seguito sono riportati alcuni esempi di ciò in azione.

Publishing to Amazon SNS

In questo esempio, il nostro resolver HTTP invia un messaggio a un argomento di Amazon SNS. L'ARN dell'argomento è noto solo dopo che lo stack che definisce l'API GraphQL e l'argomento sono stati distribuiti.

```
/**
 * Sends a publish request to the SNS topic
 */
export function request(ctx) {
  const TOPIC_ARN = ctx.env.TOPIC_ARN;
  const { input: values } = ctx.args;
  // this custom function sends values to the SNS topic
  return publishToSNSRequest(TOPIC_ARN, values);
}
```

Transactions with DynamoDB

In questo esempio, i nomi della tabella DynamoDB sono diversi se l'API è implementata per lo staging o è già in produzione. Non è necessario modificare il codice del resolver. I valori delle variabili di ambiente vengono aggiornati in base a dove viene distribuita l'API.

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { authorId, postId } = ctx.args;
  return {
    operation: 'TransactWriteItems',
    transactItems: [
      {
        table: ctx.env.POST_TABLE,
        operation: 'PutItem',
        key: util.dynamodb.toMapValues({ postId }),
        // rest of the configuration
      },
      {
        table: ctx.env.AUTHOR_TABLE,
        operation: 'UpdateItem',
        key: util.dynamodb.toMapValues({ authorId }),
        // rest of the configuration
      },
    ],
  };
}
```

```
}
```

Configurazione dell'autorizzazione e dell'autenticazione per proteggere GraphQL APIs

AWS AppSync offre i seguenti tipi di autorizzazione per proteggere GraphQL APIs: chiavi API, Lambda, IAM, OpenID Connect e Cognito User Pools. Ciascuna opzione offre un metodo di sicurezza diverso:

1. Autorizzazione tramite chiave API: controlla la limitazione per i non autenticati APIs, fornendo una semplice opzione di sicurezza.
2. Autorizzazione Lambda: abilita la logica di autorizzazione personalizzata, spiegando in dettaglio gli input e gli output delle funzioni.
3. Autorizzazione IAM: utilizza il processo AWS di firma della versione 4 della firma, che consente un controllo granulare degli accessi tramite le policy IAM.
4. Autorizzazione OpenID Connect: si integra con i servizi conformi a OIDC per l'autenticazione degli utenti.
5. Pool di utenti Cognito: implementa il controllo degli accessi basato sul gruppo utilizzando le funzionalità di gestione degli utenti di Cognito.

Tipi di autorizzazione

Esistono cinque modi per autorizzare le applicazioni a interagire con l'API AWS AppSync GraphQL. È possibile specificare il tipo di autorizzazione da utilizzare specificando uno dei seguenti valori del tipo di autorizzazione nella chiamata AWS AppSync API o CLI:

- **API_KEY**

Per usare chiavi API.

- **AWS_LAMBDA**

Per usare una AWS Lambda funzione.

- **AWS_IAM**

Per l'utilizzo delle autorizzazioni AWS Identity and Access Management ([IAM](#)).

- **OPENID_CONNECT**

Per usare il provider OpenID Connect.

- **AMAZON_COGNITO_USER_POOLS**

Per l'utilizzo di un pool di utenti Amazon Cognito.

Questi tipi di autorizzazione di base funzionano per la maggior parte degli sviluppatori. Per casi d'uso più avanzati, puoi aggiungere modalità di autorizzazione aggiuntive tramite la console, la CLI e AWS CloudFormation. Per le modalità di autorizzazione aggiuntive, AWS AppSync fornisce un tipo di autorizzazione che accetta i valori sopra elencati (ovvero `API_KEY`, `AWS_LAMBDA`, `AWS_IAM`, `OPENID_CONNECT`, e `AMAZON_COGNITO_USER_POOLS`).

Quando si specifica `API_KEY`, `AWS_LAMBDA`, o `AWS_IAM` come tipo di autorizzazione principale o predefinito, non è possibile specificarli nuovamente come una delle modalità di autorizzazione aggiuntive. Allo stesso modo, non è possibile duplicare `API_KEY`, `AWS_LAMBDA` o `AWS_IAM` per inserire le modalità di autorizzazione aggiuntive. Puoi utilizzare più pool di utenti Amazon Cognito e provider OpenID Connect. Tuttavia, non puoi utilizzare pool di utenti Amazon Cognito o provider OpenID Connect duplicati tra la modalità di autorizzazione predefinita e nessuna delle modalità di autorizzazione aggiuntive. Puoi specificare diversi client per il tuo pool di utenti Amazon Cognito o il provider OpenID Connect utilizzando l'espressione regolare di configurazione corrispondente.

Quando salvi le modifiche alla configurazione dell'API, AWS AppSync inizia a propagare le modifiche. Fino a quando la modifica alla configurazione non viene propagata, AWS AppSync continua a fornire i contenuti della configurazione precedente. Dopo la propagazione della modifica alla configurazione, inizia AWS AppSync immediatamente a pubblicare i contenuti in base alla nuova configurazione. Durante AWS AppSync la propagazione delle modifiche per un'API, non possiamo determinare se l'API sta servendo i tuoi contenuti sulla base della configurazione precedente o della nuova configurazione.

API_KEY autorizzazione

I non autenticati APIs richiedono una limitazione più rigorosa di quelli autenticati. APIs Un modo per controllare il throttling per gli endpoint GraphQL non autenticati è l'uso di chiavi API. Una chiave API è un valore hardcoded nell'applicazione che viene generato dal servizio AWS AppSync quando crei un endpoint GraphQL non autenticato. Puoi ruotare le chiavi API dalla console, dalla CLI o [AWS AppSync dal riferimento API](#).

Console

1. [Accedi Console di gestione AWS e apri la AppSync console.](#)
 - a. Nella APIs dashboard, scegli la tua API GraphQL.
 - b. Nella barra laterale, scegli Impostazioni.
2. In Modalità di autorizzazione predefinita, scegli la chiave API.
3. Nella tabella delle chiavi API, scegli Aggiungi chiave API.

Nella tabella verrà generata una nuova chiave API.

- Per eliminare una vecchia chiave API, seleziona la chiave API nella tabella, quindi scegli Elimina.
4. Scegli Save (Salva) nella parte inferiore della pagina.

CLI

1. Se non l'hai già fatto, configura il tuo accesso alla AWS CLI. Per ulteriori informazioni, consulta Nozioni di [base sulla configurazione](#).
2. Crea un oggetto API GraphQL eseguendo il [update-graphql-api](#) comando.

Dovrai digitare due parametri per questo particolare comando:

1. La `api-id` della tua API GraphQL.
2. La novità `name` della tua API. Puoi usare lo stesso `name`.
3. Il `authentication-type`, che sarà `API_KEY`.

Note

Esistono altri parametri come questi `Region` che devono essere configurati, ma di solito vengono utilizzati per impostazione predefinita i valori di configurazione CLI.

Un comando di esempio può avere il seguente aspetto:

```
aws appsync update-graphql-api --api-id abcdefghijklmnopqrstuvwxyz --name
TestAPI --authentication-type API_KEY
```

Verrà restituito un output nella CLI. Ecco un esempio in JSON:

```
{
  "graphqlApi": {
    "xrayEnabled": false,
    "name": "TestAPI",
    "authenticationType": "API_KEY",
    "tags": {},
    "apiId": "abcdefghijklmnopqrstuvwxyz",
    "uris": {
      "GRAPHQL": "https://s8i3kk3ufhe9034ujnv73r513e.appsync-api.us-west-2.amazonaws.com/graphql",
      "REALTIME": "wss://s8i3kk3ufhe9034ujnv73r513e.appsync-realtime-api.us-west-2.amazonaws.com/graphql"
    },
    "arn": "arn:aws:appsync:us-west-2:348581070237:apis/abcdefghijklmnopqrstuvwxyz"
  }
}
```

Le chiavi API sono configurabili per un massimo di 365 giorni e puoi estendere una data di scadenza esistente per un massimo di altri 365 giorni dalla data di scadenza. Le chiavi API sono consigliate per scopi di sviluppo o casi d'uso in cui è sicuro esporre un'API pubblica.

Nel client la chiave API viene specificata dall'intestazione `x-api-key`.

Ad esempio, se `API_KEY` è `'ABC123'`, puoi inviare una query GraphQL tramite `curl` in questo modo:

```
$ curl -XPOST -H "Content-Type:application/graphql" -H "x-api-key:ABC123" -d
'{"query": "query { movies { id } }"}' https://YOURAPPSYNCENDPOINT/graphql
```

AWS_LAMBDA autorizzazione

È possibile implementare la propria logica di autorizzazione API utilizzando una AWS Lambda funzione. È possibile utilizzare una funzione Lambda per l'autorizzatore principale o secondario, ma può esserci una sola funzione di autorizzazione Lambda per API. Quando si utilizzano le funzioni Lambda per l'autorizzazione, si applica quanto segue:

- Se l'API ha le modalità AWS_LAMBDA e l'AWS_IAM autorizzazione abilitate, la firma SigV4 non può essere utilizzata come token di autorizzazione. AWS_LAMBDA
- Se l'API ha le modalità di OPENID_CONNECT autorizzazione AWS_LAMBDA e o la modalità di AMAZON_COGNITO_USER_POOLS autorizzazione abilitata, il token OIDC non può essere utilizzato come token di autorizzazione. AWS_LAMBDA Nota che il token OIDC può essere uno schema Bearer.
- Una funzione Lambda non deve restituire più di 5 MB di dati contestuali per i resolver.

Ad esempio, se il token di autorizzazione è 'ABC123', puoi inviare una query GraphQL tramite curl come segue:

```
$ curl -XPOST -H "Content-Type:application/graphql" -H "Authorization:ABC123" -d '{ "query": "query { movies { id } }" }' https://YOURAPPSYNCENDPOINT/graphql
```

Le funzioni Lambda vengono chiamate prima di ogni query o mutazione. Il valore restituito può essere memorizzato nella cache in base all'ID API e al token di autenticazione. Quando una risposta di autorizzazione Lambda è inferiore a 1.048.576 byte, AWS AppSync memorizza nella cache la risposta per le richieste successive. Se la risposta dell'autorizzazione Lambda è uguale o superiore a 1.048.576 byte, AWS AppSync non memorizza nella cache la risposta e richiama l'autorizzazione Lambda per ogni richiesta in arrivo. Per ottimizzare le prestazioni e ridurre al minimo i costi di chiamata Lambda, consigliamo di limitare le risposte dell'autorizzazione Lambda a 1.048.576 byte. Per impostazione predefinita, la memorizzazione nella cache non è attivata, ma può essere abilitata a livello di API o impostando il valore nel valore restituito da una funzione. `ttlOverride`

Se lo si desidera, è possibile specificare un'espressione regolare che convalida i token di autorizzazione prima della chiamata della funzione. Queste espressioni regolari vengono utilizzate per verificare che un token di autorizzazione sia del formato corretto prima che la funzione venga chiamata. Qualsiasi richiesta che utilizza un token che non corrisponde a questa espressione regolare verrà rifiutata automaticamente.

Le funzioni Lambda utilizzate per l'autorizzazione richiedono l'applicazione `appsync.amazonaws.com` di una politica principale per consentire di AWS AppSync chiamarle. Questa azione viene eseguita automaticamente nella AWS AppSync console; la AWS AppSync console non rimuove la policy. Per ulteriori informazioni sull'associazione di policy alle funzioni Lambda, [consulta Politiche basate sulle risorse](#) nella Developer Guide. AWS Lambda

La funzione Lambda specificata riceverà un evento con la forma seguente:

```
{
  "authorizationToken": "ExampleAUTHtoken123123123",
  "requestContext": {
    "apiId": "aaaaaa123123123example123",
    "accountId": "111122223333",
    "requestId": "f4081827-1111-4444-5555-5cf4695f339f",
    "queryString": "mutation CreateEvent {...}\n\nquery MyQuery {...}\n",
    "operationName": "MyQuery",
    "variables": {}
  }
  "requestHeaders": {
    application request headers
  }
}
```

L'eventoggetto contiene le intestazioni che sono state inviate nella richiesta dal client dell'applicazione a. AWS AppSync

La funzione di autorizzazione deve restituire almeno `isAuthorized` un valore booleano che indica se la richiesta è autorizzata. AWS AppSync riconosce le seguenti chiavi restituite dalle funzioni di autorizzazione Lambda:

Note

Il valore per l'operazione `operationName` in the `requestContext` for a WebSocket connect è impostato su `"»DeepDish:Connect`. AWS AppSync

Elenco delle funzioni

`isAuthorized`(booleano, obbligatorio)

Un valore booleano che indica se il valore in `authorizationToken` è autorizzato a effettuare chiamate all'API GraphQL.

Se questo valore è vero, l'esecuzione dell'API GraphQL continua. Se questo valore è falso, `UnauthorizedException` viene generato un

deniedFields(elenco di stringhe, opzionale)

Un elenco dei quali viene modificato forzatamente in `null`, anche se un valore è stato restituito da un resolver.

Ogni articolo è un ARN di campo completamente qualificato sotto forma di `arn:aws:appsync:us-east-1:111122223333:apis/GraphQLApiId/types/TypeName/fields/FieldName` o in forma abbreviata di `TypeName.FieldName`. Il modulo ARN completo deve essere usato quando due APIs condividono un autorizzatore di funzione Lambda e potrebbe esserci ambiguità tra i tipi e i campi comuni tra i due. APIs

resolverContext(Oggetto JSON, opzionale)

Un oggetto JSON visibile come `$ctx.identity.resolverContext` nei modelli di resolver. Ad esempio, se un resolver restituisce la seguente struttura:

```
{
  "isAuthorized":true
  "resolverContext": {
    "banana":"very yellow",
    "apple":"very green"
  }
}
```

Il valore dei modelli `ctx.identity.resolverContext.apple` di resolver sarà `"». very green`. L'`resolverContext` oggetto supporta solo coppie chiave-valore. Le chiavi annidate non sono supportate.

Warning

La dimensione totale di questo oggetto JSON non deve superare i 5 MB.

ttlOverride(numero intero, opzionale)

Il numero di secondi per i quali la risposta deve essere memorizzata nella cache. Se non viene restituito alcun valore, viene utilizzato il valore dell'API. Se è 0, la risposta non viene memorizzata nella cache.

Le autorizzazioni Lambda hanno un timeout standard di 10 secondi, ma possono scadere prima in condizioni di traffico intenso. Ti consigliamo di progettare funzioni da eseguire nel minor tempo possibile (meno di 1 secondo) per scalare le prestazioni dell'API.

Più persone AWS AppSync APIs possono condividere una singola funzione Lambda di autenticazione. L'uso di un sistema di autorizzazione tra account non è consentito.

Quando condividete una funzione di autorizzazione tra più persone APIs, tenete presente che i nomi di campo in formato breve (*typename.fieldname*) possono inavvertitamente nascondere dei campi. Per disambiguare un campo indentedFields, è possibile specificare un campo non ambiguo ARN sotto forma di. `arn:aws:appsync:region:accountId:apis/GraphQLApiId/types/typename/fields/fieldName`

Per aggiungere una funzione Lambda come modalità di autorizzazione predefinita in: AWS AppSync Console

1. Accedi alla AWS AppSync console e vai all'API che desideri aggiornare.
2. Vai alla pagina Impostazioni della tua API.

Cambia l'autorizzazione a livello di API in: AWS Lambda

3. Scegli l'ARN Regione AWS e Lambda per autorizzare le chiamate API.

Note

La politica principale appropriata verrà aggiunta automaticamente, consentendo di AWS AppSync chiamare la funzione Lambda.

4. Facoltativamente, imposta il TTL di risposta e l'espressione regolare di convalida del token.

AWS CLI

1. Allega la seguente policy alla funzione Lambda in uso:

```
aws lambda add-permission --function-name "my-function" --statement-id "appsync"
--principal appsync.amazonaws.com --action lambda:InvokeFunction --output text
```

 Important

Se desideri che la politica della funzione sia bloccata su una singola API GraphQL, puoi eseguire questo comando:

```
aws lambda add-permission --function-name "my-function" --  
statement-id "appsync" --principal appsync.amazonaws.com --action  
lambda:InvokeFunction --source-arn "<my AppSync API ARN>" --output text
```

2. Aggiorna la tua AWS AppSync API per utilizzare la funzione Lambda ARN specificata come autorizzatore:

```
aws appsync update-graphql-api --api-id example2f0ur2oid7acexample --  
name exampleAPI --authentication-type AWS_LAMBDA --lambda-authorizer-config  
authorizerUri="arn:aws:lambda:us-east-2:111122223333:function:my-function"
```

 Note

Puoi anche includere altre opzioni di configurazione come l'espressione regolare del token.

L'esempio seguente descrive una funzione Lambda che dimostra i vari stati di autenticazione e di errore che una funzione Lambda può avere quando viene utilizzata come meccanismo di autorizzazione: AWS AppSync

```
def handler(event, context):  
    # This is the authorization token passed by the client  
    token = event.get('authorizationToken')  
    # If a lambda authorizer throws an exception, it will be treated as unauthorized.  
    if 'Fail' in token:  
        raise Exception('Purposefully thrown exception in Lambda Authorizer.')  
  
    if 'Authorized' in token and 'ReturnContext' in token:  
        return {  
            'isAuthorized': True,  
            'resolverContext': {  
                'key': 'value'
```

```
    }  
  }  
  
  # Authorized with no f  
  if 'Authorized' in token:  
    return {  
      'isAuthorized': True  
    }  
  # Partial authorization  
  if 'Partial' in token:  
    return {  
      'isAuthorized': True,  
      'deniedFields':['user.favoriteColor']  
    }  
  if 'NeverCache' in token:  
    return {  
      'isAuthorized': True,  
      'ttlOverride': 0  
    }  
  if 'Unauthorized' in token:  
    return {  
      'isAuthorized': False  
    }  
  # if nothing is returned, then the authorization fails.  
  return {}
```

Eludere le limitazioni di autorizzazione dei token SigV4 e OIDC

I seguenti metodi possono essere utilizzati per aggirare il problema dell'impossibilità di utilizzare la firma SigV4 o il token OIDC come token di autorizzazione Lambda quando sono abilitate determinate modalità di autorizzazione.

Se desideri utilizzare la firma SigV4 come token di autorizzazione Lambda quando le modalità di autorizzazione AWS_IAM e le modalità di AWS_LAMBDA autorizzazione sono abilitate per l'API, AWS AppSync procedi come segue:

- Per creare un nuovo token di autorizzazione Lambda, aggiungi and/or prefissi di suffissi casuali alla firma SigV4.
- Per recuperare la firma SigV4 originale, aggiorna la funzione Lambda rimuovendo i and/or suffissi casuali dei prefissi dal token di autorizzazione Lambda. Quindi, usa la firma SigV4 originale per l'autenticazione.

Se desideri utilizzare il token OIDC come token di autorizzazione Lambda quando la modalità di autorizzazione o le modalità di OPENID_CONNECT AWS_LAMBDA autorizzazione AMAZON_COGNITO_USER_POOLS e sono abilitate per l'API, AWS AppSync procedi come segue:

- Per creare un nuovo token di autorizzazione Lambda, aggiungi and/or prefissi di suffissi casuali al token OIDC. Il token di autorizzazione Lambda non deve contenere un prefisso dello schema Bearer.
- Per recuperare il token OIDC originale, aggiorna la funzione Lambda rimuovendo i and/or suffissi dei prefissi casuali dal token di autorizzazione Lambda. Quindi, usa il token OIDC originale per l'autenticazione.

AWS_IAM autorizzazione

Questo tipo di autorizzazione applica il [processo di AWS firma della versione 4 della firma sull'API GraphQL](#). È possibile associare policy d'accesso Identity and Access Management ([IAM](#)) a questo tipo di autorizzazione. La tua applicazione può sfruttare questa associazione utilizzando una chiave di accesso (che consiste in un ID chiave di accesso e una chiave di accesso segreta) o utilizzando credenziali temporanee di breve durata fornite da Amazon Cognito Federated Identities.

Se vuoi usare un ruolo che abbia accesso per eseguire tutte le operazioni sui dati:

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "appsync:GraphQL"
      ],
      "Resource": [
        "arn:aws:appsync:us-west-2:123456789012:apis/YourGraphQLApiId/*"
      ]
    }
  ]
}
```

Puoi trovarla `YourGraphQLApiId` dalla pagina principale dell'elenco delle API nella AppSync console, direttamente sotto il nome della tua API. In alternativa, puoi recuperarlo con l'interfaccia a riga di comando: `aws appsync list-graphql-apis`

Se vuoi limitare l'accesso solo a determinate operazioni GraphQL, puoi farlo per i campi `root Query`, `Mutation` e `Subscription`.

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "appsync:GraphQL"
      ],
      "Resource": [
        "arn:aws:appsync:us-west-2:123456789012:apis/YourGraphQLApiId/types/Query/fields/<Field-1>",
        "arn:aws:appsync:us-west-2:123456789012:apis/YourGraphQLApiId/types/Query/fields/<Field-2>",
        "arn:aws:appsync:us-west-2:123456789012:apis/YourGraphQLApiId/types/Mutation/fields/<Field-1>",
        "arn:aws:appsync:us-west-2:123456789012:apis/YourGraphQLApiId/types/Subscription/fields/<Field-1>"
      ]
    }
  ]
}
```

Ad esempio, supponiamo la presenza dello schema seguente e di voler limitare l'accesso in modo da ottenere tutti i post:

```
schema {
  query: Query
  mutation: Mutation
}

type Query {
  posts: [Post!]!
```

```
}

type Mutation {
  addPost(id:ID!, title:String!):Post!
}
```

La politica IAM corrispondente per un ruolo (che puoi collegare a un pool di identità di Amazon Cognito, ad esempio) sarebbe simile alla seguente:

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "appsync:GraphQL"
      ],
      "Resource": [
        "arn:aws:appsync:us-west-2:123456789012:apis/YourGraphQLApiId/
types/Query/fields/posts"
      ]
    }
  ]
}
```

OPENID_CONNECT autorizzazione

Questo tipo di autorizzazione applica i token [OpenID connect](#) (OIDC) forniti da un servizio conforme a OIDC. La tua applicazione può usare gli utenti e i privilegi definiti dal provider OIDC per controllare l'accesso.

Un URL dell'emittente è l'unico valore di configurazione necessario da fornire ad AWS AppSync, ad esempio <https://auth.example.com>. Questo URL deve essere indirizzabile tramite HTTPS. AWS AppSync [aggiunge /.well-known/openid-configuration all'URL dell'emittente e individua la configurazione OpenID in base alla specifica OpenID Connect https://auth.example.com/.well-known/openid-configuration Discovery](#). Si aspetta di recuperare un [RFC5785](#) documento JSON conforme a questo URL. Questo documento JSON deve

contenere una `jwks_uri` chiave che rimanda al documento JSON Web Key Set (JWKS) con le chiavi di firma. AWS AppSync richiede che JWKS contenga campi JSON di e. `key id`

AWS AppSync supporta un'ampia gamma di algoritmi di firma.

Algoritmi di firma

RS256

RS384

RS512

PS256

PS384

PS512

HS256

HS384

HS512

ES256

ES384

ES512

Si consiglia di utilizzare gli algoritmi RSA. I token emessi dal provider devono includere la data e l'ora di emissione del token (`iat`) e possono includere la data e l'ora in cui è stato autenticato (`auth_time`). Puoi specificare valori TTL per la data e l'ora di emissione (`iatTTL`) e di autenticazione (`authTTL`) nella configurazione OpenID Connect per un'ulteriore convalida. Se il provider autorizza più applicazioni, puoi anche immettere un'espressione regolare (`clientId`), usata per l'autorizzazione tramite ID client. Quando `clientId` è presente nella configurazione di OpenID Connect, AWS AppSync convalida l'affermazione richiedendo che corrisponda `clientId` all'affermazione `aud` o nel token.

Per convalidare più client, IDs usa l'operatore pipeline («|») che è un «o» nell'espressione regolare. Ad esempio, se l'applicazione OIDC ha quattro client con client IDs come 0A1S2D, 1F4G9H, 1J6L4B, 6GS5MG, per convalidare solo i primi tre client IDs, è necessario inserire GS5 1F4G9H|1J6L4B|6MG nel campo ID client.

Se un'API è configurata con più tipi di autorizzazione, AWS AppSync convalida l'emittente (iss claim) presente nel token JWT dalle intestazioni della richiesta confrontandolo con l'URL dell'emittente specificato nella configurazione dell'API. Tuttavia, quando un'API è configurata solo con OPENID_CONNECT autorizzazione, AWS AppSync salta questa fase di convalida dell'URL dell'emittente.

AMAZON_COGNITO_USER_POOLS autorizzazione

Questo tipo di autorizzazione applica i token OIDC forniti dai pool di utenti di Amazon Cognito. L'applicazione può sfruttare gli utenti e i gruppi presenti nei pool di utenti e nei pool di utenti di un altro AWS account e associarli ai campi GraphQL per controllare l'accesso.

Quando usi i pool di utenti di Amazon Cognito, puoi creare gruppi a cui appartengono gli utenti. Queste informazioni sono codificate in un token JWT a cui l'applicazione invia AWS AppSync in un'intestazione di autorizzazione durante l'invio di operazioni GraphQL. Puoi usare direttive GraphQL nello schema per controllare quali gruppi possono richiamare resolver specifici in un campo, per offrire un accesso più controllato ai tuoi clienti.

Ad esempio, supponiamo lo schema GraphQL seguente:

```
schema {
  query: Query
  mutation: Mutation
}

type Query {
  posts:[Post!]!
}

type Mutation {
  addPost(id:ID!, title:String!):Post!
}
...
```

Se hai due gruppi nei pool di utenti di Amazon Cognito, blogger e lettori, e desideri limitare i lettori in modo che non possano aggiungere nuove voci, lo schema dovrebbe essere simile al seguente:

```
schema {
  query: Query
  mutation: Mutation
}
```

```
type Query {
  posts:[Post!]!
  @aws_auth(cognito_groups: ["Bloggers", "Readers"])
}

type Mutation {
  addPost(id:ID!, title:String!):Post!
  @aws_auth(cognito_groups: ["Bloggers"])
}

...
```

Tieni presente che puoi omettere la `@aws_auth` direttiva se desideri impostare di default una `grant-or-deny` strategia di accesso specifica. È possibile specificare la `grant-or-deny` strategia nella configurazione del pool di utenti quando si crea l'API GraphQL tramite la console o tramite il seguente comando CLI:

```
$ aws appsync --region us-west-2 create-graphql-api --authentication-
type AMAZON_COGNITO_USER_POOLS --name userpoolstest --user-pool-config
'{"userPoolId":"test", "defaultEffect":"ALLOW", "awsRegion":"us-west-2"}'
```

Utilizzo di modalità di autorizzazione aggiuntive

Quando aggiungi modalità di autorizzazione aggiuntive, puoi configurare direttamente l'impostazione di autorizzazione a livello di API AWS AppSync GraphQL (ovvero, il `authenticationType` campo che puoi configurare direttamente sull'`GraphQLApi` oggetto) e funge da impostazione predefinita sullo schema. Ciò significa che qualsiasi tipo che non dispone di una direttiva specifica deve superare l'impostazione di autorizzazione a livello di API.

A livello di schema, puoi specificare ulteriori modalità di autorizzazione utilizzando le direttive sullo schema. Puoi specificare le modalità di autorizzazione su singoli campi nello schema. Ad esempio, per l'autorizzazione `API_KEY` puoi utilizzare `@aws_api_key` nelle definizioni/campi del

tipo di oggetto dello schema. Le direttive seguenti sono supportate nei campi dello schema e nelle definizioni del tipo di oggetto:

- `@aws_api_key` - Per specificare che il campo è autorizzato `API_KEY`.
- `@aws_iam` - Per specificare che il campo è autorizzato `AWS_IAM`.
- `@aws_oidc` - Per specificare che il campo è autorizzato `OPENID_CONNECT`.
- `@aws_cognito_user_pools` - Per specificare che il campo è autorizzato `AMAZON_COGNITO_USER_POOLS`.
- `@aws_lambda` - Per specificare che il campo è autorizzato `AWS_LAMBDA`.

Non puoi utilizzare la direttiva `@aws_auth` insieme a modalità di autorizzazione aggiuntive. `@aws_auth` funziona solo nel contesto dell'autorizzazione `AMAZON_COGNITO_USER_POOLS` senza modalità di autorizzazione aggiuntive. Tuttavia, puoi utilizzare la direttiva `@aws_cognito_user_pools` al posto della direttiva `@aws_auth`, utilizzando gli stessi argomenti. La differenza principale tra le due è che è possibile specificare `@aws_cognito_user_pools` in qualsiasi definizione di campo e tipo di oggetto.

Per comprendere come funzionano le modalità di autorizzazione aggiuntive e come possono essere specificate in uno schema, esaminiamo lo schema seguente:

```
schema {  
  query: Query  
  mutation: Mutation  
}  
  
type Query {  
  getPost(id: ID!): Post  
  getAllPosts(): [Post]  
  @aws_api_key  
}  
  
type Mutation {  
  addPost(  
    id: ID!  
    author: String!  
    title: String!  
    content: String!  
    url: String!  
  ): Post!
```

```
}

type Post @aws_api_key @aws_iam {
  id: ID!
  author: String
  title: String
  content: String
  url: String
  ups: Int!
  downs: Int!
  version: Int!
}

...
```

Per questo schema, supponiamo che `AWS_IAM` sia il tipo di autorizzazione predefinito sull'API AWS AppSync GraphQL. Ciò significa che i campi che non dispongono di una direttiva sono protetti utilizzando `AWS_IAM`. Ad esempio, questo è il caso del campo `getPost` sul tipo `Query`. Le direttive dello schema consentono di utilizzare più di una modalità di autorizzazione. Ad esempio, è possibile `API_KEY` configurarla come modalità di autorizzazione aggiuntiva sull'API AWS AppSync GraphQL e contrassegnare un campo utilizzando la `@aws_api_key` direttiva (ad esempio, `getAllPosts` in questo esempio). Le direttive funzionano a livello di campo, quindi è necessario concedere a `API_KEY` l'accesso anche al tipo `Post`. Puoi eseguire questa operazione contrassegnando ogni campo nel tipo `Post` con una direttiva oppure contrassegnando il tipo `Post` con la direttiva `@aws_api_key`.

Per limitare ulteriormente l'accesso ai campi nel tipo `Post`, puoi utilizzare le direttive sui singoli campi nel tipo `Post` come mostrato di seguito.

Ad esempio, puoi aggiungere un campo `restrictedContent` al tipo `Post` e limitare l'accesso utilizzando la direttiva `@aws_iam`. Le richieste autenticate `AWS_IAM` possono accedere a `restrictedContent`, mentre le richieste `API_KEY` non possono accedervi.

```
type Post @aws_api_key @aws_iam{
  id: ID!
  author: String
  title: String
  content: String
  url: String
  ups: Int!
  downs: Int!
  version: Int!
  restrictedContent: String @aws_iam
}
```



```
restrictedContent: String!  
  @aws_iam  
}  
...
```

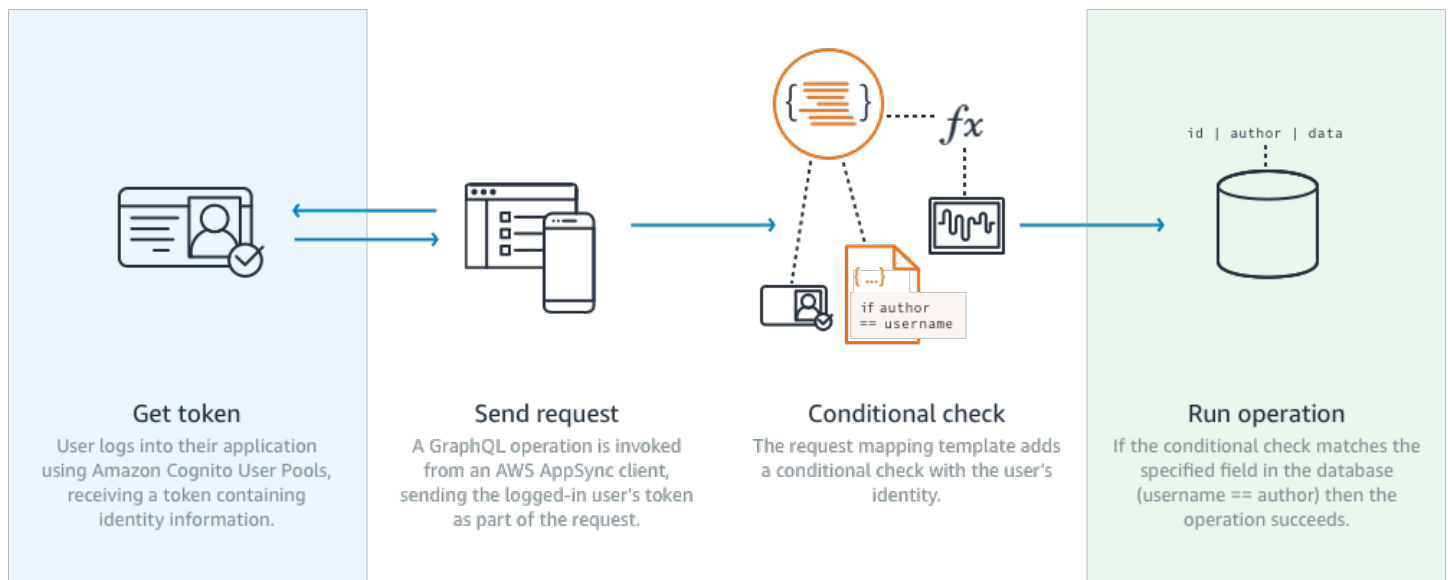
Controllo granulare degli accessi

Le informazioni precedenti descrivono come limitare o concedere l'accesso a determinati campi GraphQL. Se vuoi impostare controlli degli accessi sui dati in base a determinate condizioni (ad esempio in base all'utente che effettua una chiamata e se è il proprietario dei dati), puoi usare i modelli di mappatura nei resolver. Puoi anche eseguire una logica di business più complessa, che descriveremo in [Applicazione di filtri alle informazioni](#).

Questa sezione mostra come impostare i controlli di accesso sui dati utilizzando un modello di mappatura del resolver DynamoDB.

Prima di procedere oltre, se non hai dimestichezza con i modelli di mappatura di Resolver AWS AppSync, puoi consultare il riferimento al modello di mappatura [Resolver e il riferimento al modello di mappatura Resolver](#) per DynamoDB.

Nell'esempio seguente che utilizza DynamoDB, supponiamo di utilizzare lo schema di post del blog precedente e che solo gli utenti che hanno creato un post siano autorizzati a modificarlo. Il processo di valutazione ha lo scopo di permettere all'utente di ottenere le credenziali nell'applicazione, ad esempio usando pool di utenti Amazon Cognito, e quindi passare queste credenziali come parte di un'operazione GraphQL. Il modello di mappatura sostituisce quindi un valore delle credenziali, ad esempio il nome utente, in un'istruzione condizionale, che viene quindi confrontata con un valore nel database.



Per aggiungere questa funzionalità, aggiungi un campo GraphQL `editPost` in questo modo:

```
schema {
  query: Query
  mutation: Mutation
}

type Query {
  posts:[Post!]!
}

type Mutation {
  editPost(id:ID!, title:String, content:String):Post
  addPost(id:ID!, title:String!):Post!
}
...
```

Il modello di mappatura dei resolver per `editPost` (mostrato in un esempio alla fine di questa sezione) deve eseguire un controllo logico sul datastore per permettere solo all'utente che ha creato un post di modificarlo. Poiché si tratta di un'operazione di modifica, corrisponde a un'operazione `UpdateItem` in DynamoDB. Puoi eseguire un controllo condizionale prima di eseguire questa operazione, usando il contesto passato per la convalida dell'identità dell'utente. Questo viene archiviato in un oggetto `Identity` che ha i valori seguenti:

```
{
  "accountId" : "12321434323",
```

```
"cognitoIdentityPoolId" : "",
"cognitoIdentityId" : "",
"sourceIP" : "",
"caller" : "ThisistheprincipalARN",
"username" : "username",
"userArn" : "Sameasabove"
}
```

Per utilizzare questo oggetto in una chiamata `UpdateItem` DynamoDB, è necessario memorizzare le informazioni sull'identità dell'utente nella tabella per il confronto. Prima di tutto, la mutazione `addPost` deve archiviare l'autore. In secondo luogo, la mutazione `editPost` deve eseguire il controllo condizionale prima dell'aggiornamento.

Ecco un esempio di codice del resolver `addPost` che memorizza l'identità dell'utente come colonna: `Author`

```
import { util, Context } from '@aws-appsync/utils';
import { put } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  const { id: postId, ...item } = ctx.args;
  return put({
    key: { postId },
    item: { ...item, Author: ctx.identity.username },
    condition: { postId: { attributeExists: false } },
  });
}

export const response = (ctx) => ctx.result;
```

Nota che l'attributo `Author` viene popolato dall'oggetto `Identity`, che proviene dall'applicazione.

Infine, ecco un esempio del codice resolver `foreditPost`, che aggiorna il contenuto del post sul blog solo se la richiesta proviene dall'utente che ha creato il post:

```
import { util, Context } from '@aws-appsync/utils';
import { put } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  const { id, ...item } = ctx.args;
  return put({
```

```
key: { id },
item,
condition: { author: { contains: ctx.identity.username } },
});
}

export const response = (ctx) => ctx.result;
```

Questo esempio utilizza un `PutItem` che sovrascrive tutti i valori anziché uno `UpdateItem`, ma lo stesso concetto si applica al `condition` blocco di istruzioni.

Filtraggio delle informazioni

Esistono alcuni casi in cui non puoi controllare la risposta proveniente dall'origine dati, ma non vuoi inviare informazioni inutili ai client in un'operazione di scrittura o lettura riuscita nell'origine dati. In questi casi, puoi filtrare le informazioni usando un modello di mappatura della risposta.

Ad esempio, supponiamo di non avere un indice appropriato nella tabella DynamoDB del post sul blog (ad esempio un indice su). `Author` Puoi usare il seguente resolver:

```
import { util, Context } from '@aws-appsync/utils';
import { get } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  return get({ key: { ctx.args.id } });
}

export function response(ctx) {
  if (ctx.result.author === ctx.identity.username) {
    return ctx.result;
  }
  return null;
}
```

Il gestore della richiesta recupera l'elemento anche se il chiamante non è l'autore che ha creato il post. Per evitare che ciò restituisca tutti i dati, il gestore della risposta verifica che il chiamante corrisponda all'autore dell'elemento. Se il chiamante non corrisponde a questo controllo, viene restituita solo una risposta `Null`.

Accesso origine dati

AWS AppSync comunica con le fonti di dati utilizzando i ruoli e le policy di accesso di Identity and Access Management ([IAM](#)). Se si utilizza un ruolo esistente, è necessario aggiungere una politica di fiducia per AWS AppSync assumere il ruolo. La relazione di attendibilità apparirà come segue:

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "appsync.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

È importante ridurre l'ambito della policy di accesso sul ruolo per avere solo le autorizzazioni per agire sul set minimo di risorse necessarie. Quando si utilizza la AppSync console per creare un'origine dati e creare un ruolo, questa operazione viene eseguita automaticamente. Tuttavia, quando si utilizza un modello di esempio integrato dalla console IAM per creare un ruolo al di fuori della console AWS AppSync, non sarà automaticamente ridotto l'ambito dei permessi su una risorsa; si consiglia di eseguire questa azione prima di spostare l'applicazione in produzione.

Casi d'uso del controllo degli accessi per proteggere richieste e risposte

Nella sezione [Sicurezza](#) sono stati illustrati i diversi metodi di autorizzazione per proteggere l'API ed è stato presentato il meccanismo di autorizzazione granulare per comprendere i concetti e il flusso. Poiché AWS AppSync consente di eseguire operazioni logiche complete sui dati tramite l'uso di [modelli GraphQL Resolver Mapping](#), è possibile proteggere i dati in lettura o scrittura in modo molto flessibile utilizzando una combinazione di identità utente, condizionali e data injection.

[Se non hai dimestichezza con la modifica dei AWS AppSync Resolver, consulta la guida alla programmazione.](#)

Panoramica di

La concessione dell'accesso ai dati in un sistema viene tradizionalmente effettuata tramite una [matrice di controllo degli accessi](#) in cui l'intersezione di una riga (risorsa) e una colonna (utente/ruolo) rappresenta le autorizzazioni concesse.

AWS AppSync utilizza le risorse del proprio account e inserisce le informazioni sull'identità (utente/ruolo) nella richiesta e risposta GraphQL come [oggetto di contesto](#), che può essere utilizzato nel resolver. Ciò significa che le autorizzazioni possono essere concesse in modo appropriato per le operazioni di lettura o scrittura in base alla logica del resolver. Se questa logica è a livello di risorsa, ad esempio solo determinati utenti o gruppi denominati possono accedere read/write a una riga specifica del database, è necessario archiviare tali «metadati di autorizzazione». AWS AppSync non memorizza alcun dato, quindi è necessario archiviare questi metadati di autorizzazione con le risorse in modo da poter calcolare le autorizzazioni. I metadati di autorizzazione sono in genere un attributo (colonna) in una tabella DynamoDB, ad esempio un proprietario o un elenco di utenti/gruppi. Potrebbero ad esempio esserci gli attributi Readers e Writers.

Da un punto di vista generale, ciò significa che se stai leggendo una singola voce da un'origine dati, esegui un'istruzione `#if ( ) ... #end` condizionale nel modello di risposta dopo che il resolver ha letto dall'origine dati. Il controllo usa in genere i valori di utenti o gruppi in `$context.identity` per i controlli di appartenenza in base ai metadati di autorizzazione restituiti da un'operazione di lettura. Per più record, ad esempio gli elenchi restituiti da una tabella Scan o Query, puoi inviare il controllo della condizione come parte dell'operazione all'origine dati usando valori di utenti o gruppi simili.

Analogamente, quando scrivi dati applichi un'istruzione condizionale all'operazione (ad esempio PutItem o UpdateItem) per verificare se l'utente o il gruppo che esegue la mutazione dispone di autorizzazione. Molto spesso l'istruzione condizionale usa un valore in `$context.identity` per eseguire il confronto in base ai metadati di autorizzazione nella risorsa. Per entrambi i modelli di richiesta e di risposta è anche possibile usare intestazioni personalizzate provenienti dai client per eseguire i controlli di convalida.

Lettura dei dati

Come illustrato in precedenza, i metadati di autorizzazione per eseguire un controllo devono essere archiviati con una risorsa o passati nella richiesta GraphQL (identità, intestazione e così via). Per illustrare questo concetto, supponi di avere la tabella DynamoDB seguente:

ID	Data	PeopleCanAccess	GroupsCanAccess	Owner
123	{my: data,...}	[Mary, Joe]	[Admins, Editors]	Nadia

La chiave primaria è `id` e i dati a cui accedere corrispondono a `Data`. Le altre colonne sono esempi di controlli che è possibile eseguire per l'autorizzazione. `Owner` richiederebbe un `String` po' di tempo `PeopleCanAccess` e `GroupsCanAccess` sarebbe `String Sets` come indicato nel [riferimento del modello di mappatura Resolver](#) per DynamoDB.

Nella [panoramica sui modelli di mappatura dei resolver](#) il diagramma mostra che il modello di risposta contiene non solo l'oggetto `context`, ma anche i risultati dall'origine dati. Per le query GraphQL di singole voci, è possibile usare il modello di risposta per controllare se l'utente è autorizzato a visualizzare i risultati oppure restituire un messaggio di errore di autorizzazione. In questo caso si parla talvolta di "filtro di autorizzazione". Per le query GraphQL che restituiscono elenchi, usando un elemento `Scan` o `Query`, è preferibile eseguire il controllo nel modello di richiesta e restituire i dati solo se la condizione di autorizzazione viene soddisfatta. L'implementazione è quindi la seguente:

1. `GetItem` - controllo delle autorizzazioni per i singoli record. Eseguito usando istruzioni `#if()` ... `#end`.
2. Operazioni `Scan/Query`: il controllo di autorizzazione è un'istruzione `"filter"`: `{"expression": ...}`. I controlli comuni riguardano l'uguaglianza (`attribute = :input`) o la verifica della presenza di un valore in un elenco (`contains(attribute, :input)`).

Nel punto 2 l'elemento `attribute` in entrambe le istruzioni rappresenta il nome di colonna del record in una tabella, come `Owner` nell'esempio precedente. Puoi impostare un alias con un segno `#` e usare `"expressionNames": {...}`, ma non è obbligatorio. L'elemento `:input` è un riferimento al valore che stai confrontando con l'attributo di database, che definirai in `"expressionValues": {...}`. Gli esempi sono disponibili di seguito.

Caso d'uso: il proprietario può leggere

Usando la tabella precedente, se desideri restituire i dati solo se `Owner == Nadia` per una singola operazione di lettura (`GetItem`), il modello sarà analogo al seguente:

```
#if($context.result["Owner"] == $context.identity.username)
  $utils.toJson($context.result)
```

```
#else
  $utils.unauthorized()
#end
```

Ci sono alcuni aspetti da sottolineare che valgono anche per le sezioni successive. Innanzitutto, il controllo utilizza `$context.identity.username` il nome di registrazione intuitivo dell'utente se vengono utilizzati i pool di utenti di Amazon Cognito e l'identità dell'utente se viene utilizzato IAM (incluse le identità federate di Amazon Cognito). Esistono altri valori da memorizzare per un proprietario, come il valore univoco «Amazon Cognito identity», utile per federare gli accessi da più sedi, e dovresti esaminare le opzioni disponibili nel [Resolver Mapping Template Context Reference](#).

In secondo luogo, il controllo `else` condizionale che risponde con `$util.unauthorized()` è completamente facoltativo, ma consigliato come best practice nella progettazione dell'API GraphQL.

Caso d'uso: accesso specifico per codice rigido

```
// This checks if the user is part of the Admin group and makes the call
#foreach($group in $context.identity.claims.get("cognito:groups"))
  #if($group == "Admin")
    #set($inCognitoGroup = true)
  #end
#end
#if($inCognitoGroup)
{
  "version" : "2017-02-28",
  "operation" : "UpdateItem",
  "key" : {
    "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
  },
  "attributeValues" : {
    "owner" : $util.dynamodb.toDynamoDBJson($context.identity.username)
    #foreach( $entry in $context.arguments.entrySet() )
      , "{$entry.key}" : $util.dynamodb.toDynamoDBJson($entry.value)
    #end
  }
}
#else
  $utils.unauthorized()
#end
```


Caso d'uso: filtraggio di un elenco di risultati

Nell'esempio precedente è stato eseguito un controllo direttamente su `$context.result`, in quanto è stata restituita una singola voce, tuttavia alcune operazioni come una scansione restituiscono più voci in `$context.result.items`, quindi è necessario applicare il filtro di autorizzazione per restituire solo i risultati che l'utente può vedere. Supponiamo che questa volta `Owner` nel campo sia impostato l'`IdentityID` di Amazon Cognito nel record, quindi puoi utilizzare il seguente modello di mappatura delle risposte per filtrare in modo da mostrare solo i record di proprietà dell'utente:

```
#set($myResults = [])
#foreach($item in $context.result.items)
  ##For userpools use $context.identity.username instead
  #if($item.Owner == $context.identity.cognitoIdentityId)
    #set($added = $myResults.add($item))
  #end
#end
$utils.toJson($myResults)
```

Caso d'uso: più persone possono leggere

Un'altra opzione di autorizzazione comune consiste nel permettere a un gruppo di persone di leggere i dati. Nell'esempio seguente `"filter":{"expression":...}` restituisce i valori di una scansione di tabella solo se l'utente che esegue la query GraphQL è incluso nel set `PeopleCanAccess`.

```
{
  "version" : "2017-02-28",
  "operation" : "Scan",
  "limit": #if(${context.arguments.count}) $util.toJson($context.arguments.count)
#else 20 #end,
  "nextToken": #if(${context.arguments.nextToken})
$util.toJson($context.arguments.nextToken) #else null #end,
  "filter":{
    "expression": "contains(#peopleCanAccess, :value)",
    "expressionNames": {
      "#peopleCanAccess": "peopleCanAccess"
    },
    "expressionValues": {
      ":value": $util.dynamodb.toDynamoDBJson($context.identity.username)
    }
  }
}
```

Caso d'uso: il gruppo può leggere

Analogamente all'ultimo caso d'uso, è possibile che solo le persone in uno o più gruppi abbiano i diritti per leggere determinate voci in un database. L'uso dell'operazione "expression": "contains()" è analogo, tuttavia nell'appartenenza a un set è necessario tenere conto dell'operatore logico OR che permette di includere tutti i gruppi di cui un utente potrebbe far parte. In questo caso, creiamo l'istruzione \$expression seguente per ogni gruppo a cui appartiene l'utente e la passiamo al filtro:

```
#set($expression = "")
#set($expressionValues = {})
#foreach($group in $context.identity.claims.get("cognito:groups"))
    #set( $expression = "${expression} contains(groupsCanAccess, :var
$foreach.count )" )
    #set( $val = {})
    #set( $test = $val.put("S", $group))
    #set( $values = $expressionValues.put(":var$foreach.count", $val))
    #if ( $foreach.hasNext )
    #set( $expression = "${expression} OR" )
    #end
#end
{
    "version" : "2017-02-28",
    "operation" : "Scan",
    "limit": #if(${context.arguments.count}) $util.toJson($context.arguments.count)
#else 20 #end,
    "nextToken": #if(${context.arguments.nextToken})
$util.toJson($context.arguments.nextToken) #else null #end,
    "filter":{
        "expression": "$expression",
        "expressionValues": $utils.toJson($expressionValues)
    }
}
```

Scrittura di dati

La scrittura di dati nelle mutazioni è sempre controllata nel modello di mappatura della richiesta. Nel caso delle origini dati DynamoDB, la chiave consiste nell'usare un elemento "condition": {"expression"...} appropriato che esegue la convalida in base ai metadati di autorizzazione nella tabella. In [Sicurezza](#), abbiamo fornito un esempio che è possibile utilizzare per controllare il campo Author in una tabella. In questa sezione vengono esaminati altri casi d'uso.

Caso d'uso: più proprietari

Usando il diagramma della tabella di esempio precedente, presupponi l'elenco PeopleCanAccess:

```
{
  "version" : "2017-02-28",
  "operation" : "UpdateItem",
  "key" : {
    "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
  },
  "update" : {
    "expression" : "SET meta = :meta",
    "expressionValues": {
      ":meta" : $util.dynamodb.toDynamoDBJson($ctx.args.meta)
    }
  },
  "condition" : {
    "expression" : "contains(Owner,:expectedOwner)",
    "expressionValues" : {
      ":expectedOwner" :
$util.dynamodb.toDynamoDBJson($context.identity.username)
    }
  }
}
```

Caso d'uso: il gruppo può creare un nuovo record

```
#set($expression = "")
#set($expressionValues = {})
#foreach($group in $context.identity.claims.get("cognito:groups"))
  #set( $expression = "${expression} contains(groupsCanAccess, :var
$foreach.count )" )
  #set( $val = {})
  #set( $test = $val.put("S", $group))
  #set( $values = $expressionValues.put(":var$foreach.count", $val))
  #if ( $foreach.hasNext )
    #set( $expression = "${expression} OR" )
  #end
#end
{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key" : {
```

```

    ## If your table's hash key is not named 'id', update it here. **
    "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    ## If your table has a sort key, add it as an item here. **
  },
  "attributeValues" : {
    ## Add an item for each field you would like to store to Amazon DynamoDB. **
    "title" : $util.dynamodb.toDynamoDBJson($ctx.args.title),
    "content": $util.dynamodb.toDynamoDBJson($ctx.args.content),
    "owner": $util.dynamodb.toDynamoDBJson($context.identity.username)
  },
  "condition" : {
    "expression": $util.toJson("attribute_not_exists(id) AND $expression"),
    "expressionValues": $utils.toJson($expressionValues)
  }
}

```

Caso d'uso: il gruppo può aggiornare il record esistente

```

#set($expression = "")
#set($expressionValues = {})
#foreach($group in $context.identity.claims.get("cognito:groups"))
  #set( $expression = "${expression} contains(groupsCanAccess, :var
$foreach.count )" )
  #set( $val = {})
  #set( $test = $val.put("S", $group))
  #set( $values = $expressionValues.put(":var$foreach.count", $val))
  #if ( $foreach.hasNext )
  #set( $expression = "${expression} OR" )
  #end
#end
{
  "version" : "2017-02-28",
  "operation" : "UpdateItem",
  "key" : {
    "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
  },
  "update":{
    "expression" : "SET title = :title, content = :content",
    "expressionValues": {
      ":title" : $util.dynamodb.toDynamoDBJson($ctx.args.title),
      ":content" : $util.dynamodb.toDynamoDBJson($ctx.args.content)
    }
  },

```

```

    "condition" : {
      "expression": $util.toJson($expression),
      "expressionValues": $utils.toJson($expressionValues)
    }
  }
}

```

Record pubblici e privati

Con i filtri condizionali, è anche possibile scegliere di contrassegnare i dati come privati, pubblici o con altri valori booleani. Questa impostazione può quindi essere integrata in un filtro di autorizzazione all'interno del modello di risposta. Usando questo controllo, è possibile nascondere temporaneamente i dati o rimuoverli dalla visualizzazione senza tentare di controllare l'appartenenza a un gruppo.

Supponi, ad esempio, di aggiungere un attributo in ogni voce della tabella DynamoDB chiamata `public` con un valore `yes` o `no`. Il modello di risposta seguente può essere usato in una chiamata `GetItem` per visualizzare i dati solo nel caso in cui l'utente si trova in un gruppo che ha accesso E se i dati sono contrassegnati come pubblici:

```

#set($permissions = $context.result.GroupsCanAccess)
#set($claimPermissions = $context.identity.claims.get("cognito:groups"))

#foreach($per in $permissions)
  #foreach($cgroups in $claimPermissions)
    #if($cgroups == $per)
      #set($hasPermission = true)
    #end
  #end
#end

#if($hasPermission && $context.result.public == 'yes')
  $utils.toJson($context.result)
#else
  $utils.unauthorized()
#end

```

Nel codice precedente è anche possibile usare un operatore OR logico (`||`) per permettere agli utenti la lettura se dispongono dell'autorizzazione per un record o se il record è pubblico:

```

#if($hasPermission || $context.result.public == 'yes')
  $utils.toJson($context.result)
#else

```

```
$utils.unauthorized()  
#end
```

In generale, gli operatori standard `==`, `!=`, `&&` e `||`, sono utili per i controlli di autorizzazione.

Dati in tempo reale

È possibile applicare controlli di accesso granulari alle sottoscrizioni GraphQL nel momento in cui un client esegue una sottoscrizione, usando le stesse tecniche descritte in precedenza in questa documentazione. È possibile collegare un resolver al campo della sottoscrizione e quindi è possibile eseguire query sui dati di un'origine dati e applicare la logica condizionale nel modello di mappatura della richiesta o della risposta. Puoi anche restituire dati aggiuntivi al client, ad esempio i risultati iniziali di una sottoscrizione, a condizione che la struttura dei dati corrisponda a quella del tipo restituito nella sottoscrizione GraphQL.

Caso d'uso: l'utente può iscriversi solo a conversazioni specifiche

Un caso d'uso comune per i dati in tempo reale con le sottoscrizioni GraphQL consiste nella creazione di un'applicazione di messaggistica o di chat privata. Quando crei un'applicazione di chat con più utenti, le conversazioni possono avvenire tra due persone o tra più persone. Gli utenti possono essere raggruppati in "stanze", private o pubbliche. In questo caso, è necessario autorizzare un utente a eseguire solo la sottoscrizione di una conversazione (con una sola persona o con un gruppo) per la quale gli è stato concesso l'accesso. A scopo illustrativo, nell'esempio seguente è rappresentato un caso d'uso semplice di un utente che invia un messaggio privato a un altro utente. La configurazione ha due tabelle Amazon DynamoDB:

- Tabella Messages: (chiave primaria) `toUser`, (chiave di ordinamento) `id`
- Tabella Permissions: (chiave primaria) `username`

La tabella Messages archivia i messaggi effettivi inviati tramite una mutazione GraphQL. La tabella Permissions viene controllata dalla sottoscrizione GraphQL per verificare le autorizzazioni al momento della connessione client. L'esempio seguente presuppone che si stia usando lo schema GraphQL illustrato di seguito:

```
input CreateUserPermissionsInput {  
  user: String!  
  isAuthorizedForSubscriptions: Boolean  
}
```

```
type Message {
  id: ID
  toUser: String
  fromUser: String
  content: String
}

type MessageConnection {
  items: [Message]
  nextToken: String
}

type Mutation {
  sendMessage(toUser: String!, content: String!): Message
  createUserPermissions(input: CreateUserPermissionsInput!): UserPermissions
  updateUserPermissions(input: UpdateUserPermissionInput!): UserPermissions
}

type Query {
  getMyMessages(first: Int, after: String): MessageConnection
  getUserPermissions(user: String!): UserPermissions
}

type Subscription {
  newMessage(toUser: String!): Message
    @aws_subscribe(mutations: ["sendMessage"])
}

input UpdateUserPermissionInput {
  user: String!
  isAuthorizedForSubscriptions: Boolean
}

type UserPermissions {
  user: String
  isAuthorizedForSubscriptions: Boolean
}

schema {
  query: Query
  mutation: Mutation
  subscription: Subscription
}
```

Alcune delle operazioni standard, ad esempio, non sono illustrate di seguito per illustrare i resolver in abbonamento, ma sono implementazioni standard dei resolver DynamoDB. `createUserPermissions()` Vengono invece analizzati i flussi di autorizzazione della sottoscrizione con i resolver. Per inviare un messaggio da un utente a un altro, collega un resolver al campo `sendMessage()` e seleziona l'origine dati della tabella `Messages` con il modello di richiesta seguente:

```
{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key" : {
    "toUser" : $util.dynamodb.toDynamoDBJson($ctx.args.toUser),
    "id" : $util.dynamodb.toDynamoDBJson($util.autoId())
  },
  "attributeValues" : {
    "fromUser" : $util.dynamodb.toDynamoDBJson($context.identity.username),
    "content" : $util.dynamodb.toDynamoDBJson($ctx.args.content),
  }
}
```

In questo esempio viene utilizzato `$context.identity.username`. Ciò restituisce le informazioni sugli utenti AWS Identity and Access Management o per gli utenti di Amazon Cognito. Il modello di risposta è un passthrough semplice di `$util.toJson($ctx.result)`. Salva e torna alla pagina dello schema. Collega quindi un resolver alla sottoscrizione `newMessage()`, usando la tabella `Permissions` come origine dati e il modello di mappatura della richiesta seguente:

```
{
  "version": "2018-05-29",
  "operation": "GetItem",
  "key": {
    "username": $util.dynamodb.toDynamoDBJson($ctx.identity.username),
  },
}
```

Usa quindi il modello di mappatura della risposta seguente per eseguire i controlli di autorizzazione con i dati della tabella `Permissions`:

```
#if(! ${context.result})
  $utils.unauthorized()
#elseif(${context.identity.username} != ${context.arguments.toUser})
```



```
$utils.unauthorized()  
#elseif(! ${context.result.isAuthorizedForSubscriptions})  
    $utils.unauthorized()  
#else  
    ##User is authorized, but we return null to continue  
    null  
#end
```

In questo caso, esegui tre controlli di autorizzazione. Il primo garantisce che venga restituito un risultato. Il secondo garantisce che l'utente non esegua la sottoscrizione di messaggi destinati a un'altra persona. Il terzo assicura che l'utente sia autorizzato a sottoscrivere qualsiasi campo, controllando un attributo DynamoDB `isAuthorizedForSubscriptions` di stored as a `BOOL`.

Per testare le cose, puoi accedere alla AWS AppSync console utilizzando i pool di utenti di Amazon Cognito e un utente chiamato «Nadia», quindi eseguire il seguente abbonamento GraphQL:

```
subscription AuthorizedSubscription {  
  newMessage(toUser: "Nadia") {  
    id  
    toUser  
    fromUser  
    content  
  }  
}
```

Se nella tabella Autorizzazioni c'è un record per l'attributo chiave `username` corrispondente a `Nadia` con valore di `isAuthorizedForSubscriptions` impostato su `true`, l'operazione avrà esito positivo. Se provi a usare un valore di `username` diverso nella query `newMessage()` precedente, verrà restituito un errore.

Utilizzato AWS WAF per proteggere il tuo AWS AppSync APIs

AWS WAF è un firewall per applicazioni Web che aiuta a proteggere le applicazioni Web e APIs dagli attacchi. Consente di configurare una serie di regole, denominate elenco di controllo degli accessi Web (Web ACL), che consentono, bloccano o monitorano (contano) le richieste Web in base a regole e condizioni di sicurezza Web personalizzabili definite dall'utente. Quando integri la tua AWS AppSync API con AWS WAF, ottieni maggiore controllo e visibilità sul traffico HTTP accettato dalla tua API. Per ulteriori informazioni AWS WAF, consulta [How AWS WAF Works](#) nella AWS WAF Developer Guide.

Puoi utilizzarla AWS WAF per proteggere la tua AppSync API da exploit web comuni, come attacchi SQL injection e cross-site scripting (XSS). Questi potrebbero influire sulla disponibilità e sulle prestazioni delle API, compromettere la sicurezza o consumare risorse eccessive. Ad esempio, puoi creare regole per consentire o bloccare richieste da intervalli di indirizzi IP specificati, richieste da blocchi CIDR, richieste provenienti da un paese o una regione specifici, richieste che contengono codice SQL dannoso o richieste contenenti script dannoso.

Puoi anche creare regole che corrispondono a una stringa specificata o un modello di espressione regolare in intestazioni HTTP, metodo, stringa di query, URI e il corpo della richiesta (entro i primi 8 KB). Inoltre, puoi creare regole per bloccare attacchi da utenti-agenti, bad bot e scraper di contenuti. Ad esempio, puoi usare le regole basate sulla frequenza per specificare il numero di richieste Web consentite da ogni IP client in un periodo di 5 minuti, costantemente aggiornato, finale.

[Per ulteriori informazioni sui tipi di regole supportate e sulle AWS WAF funzionalità aggiuntive, consulta la AWS WAF Developer Guide e l'API Reference.](#)

Important

AWS WAF è la tua prima linea di difesa contro gli exploit web. Quando AWS WAF è abilitata su un'API, AWS WAF le regole vengono valutate prima di altre funzionalità di controllo degli accessi, come l'autorizzazione delle chiavi API, le policy IAM, i token OIDC e i pool di utenti di Amazon Cognito.

Integra un'API con AppSync AWS WAF

Puoi integrare un'API Appsync AWS WAF utilizzando il Console di gestione AWS AWS CLI AWS CloudFormation, o qualsiasi altro client compatibile.

Per integrare un' AWS AppSync API con AWS WAF

1. Crea un ACL AWS WAF web. Per i passaggi dettagliati sull'utilizzo della [AWS WAF console](#), consulta [Creazione di un ACL Web](#).
2. Definisci le regole per l'ACL web. Una o più regole vengono definite nel processo di creazione dell'ACL Web. Per informazioni su come strutturare le regole, consulta [AWS WAF regole](#). Per esempi di regole utili che puoi definire per la tua AWS AppSync API, consulta [Creazione di regole per un ACL web](#).

3. Associa l'ACL web a un' AWS AppSync API. È possibile eseguire questo passaggio nella [AWS WAF console](#) o nella [AppSync console](#).
- Per associare l'ACL Web a un' AWS AppSync API nella AWS WAF console, segui le istruzioni per [associare o dissociare un ACL Web a una AWS risorsa nella Developer Guide](#). AWS WAF
 - Per associare l'ACL Web a un'API nella console AWS AppSync AWS AppSync
 - a. Accedi Console di gestione AWS e apri la [AppSync console](#).
 - b. Scegli l'API che desideri associare a un ACL web.
 - c. Nel pannello di navigazione scegli Impostazioni.
 - d. Nella sezione Web Application Firewall, attiva Abilita. AWS WAF
 - e. Nell'elenco a discesa Web ACL, scegli il nome dell'ACL Web da associare alla tua API.
 - f. Scegli Salva per associare l'ACL web alla tua API.

Note

Dopo aver creato un ACL Web nella AWS WAF Console, possono essere necessari alcuni minuti prima che il nuovo ACL Web sia disponibile. Se non vedi un ACL web appena creato nel menu Web Application Firewall, attendi qualche minuto e riprova i passaggi per associare l'ACL web alla tua API.

Note

AWS WAF l'integrazione supporta l'`Subscription registration messageevent` solo per gli endpoint in tempo reale. AWS AppSync risponderà con un messaggio di errore anziché con un `start_ack` messaggio per qualsiasi persona `Subscription registration message` bloccata da AWS WAF.

Dopo aver associato un ACL Web a un' AWS AppSync API, gestirai l'ACL Web utilizzando. AWS WAF APIs Non è necessario associare nuovamente l'ACL Web all'API, a meno che non si desideri associare l' AWS AppSync AWS AppSync API a un ACL Web diverso.

Creazione di regole per un ACL web

Le regole definiscono come esaminare le richieste Web e cosa fare quando una richiesta Web soddisfa i criteri di ispezione. Le regole non esistono di AWS WAF per sé. È possibile accedere a una regola per nome in un gruppo di regole o nell'ACL Web in cui è definita. Per ulteriori informazioni, consulta [AWS WAF le regole](#). Gli esempi seguenti mostrano come definire e associare regole utili per proteggere un' AppSync API.

Example regola web ACL per limitare la dimensione del corpo della richiesta

Di seguito è riportato un esempio di regola che limita la dimensione del corpo delle richieste. Questo verrebbe inserito nell'editor Rule JSON durante la creazione di un ACL web nella AWS WAF console.

```
{
  "Name": "BodySizeRule",
  "Priority": 1,
  "Action": {
    "Block": {}
  },
  "Statement": {
    "SizeConstraintStatement": {
      "ComparisonOperator": "GE",
      "FieldToMatch": {
        "Body": {}
      },
      "Size": 1024,
      "TextTransformations": [
        {
          "Priority": 0,
          "Type": "NONE"
        }
      ]
    }
  },
  "VisibilityConfig": {
    "CloudWatchMetricsEnabled": true,
    "MetricName": "BodySizeRule",
    "SampledRequestsEnabled": true
  }
}
```

Dopo aver creato l'ACL Web utilizzando la regola di esempio precedente, è necessario associarlo all'API. AppSync In alternativa all'utilizzo di Console di gestione AWS, è possibile eseguire questo passaggio AWS CLI eseguendo il comando seguente.

```
aws waf associate-web-acl --web-acl-id waf-web-acl-arn --resource-arn appsync-api-arn
```

La propagazione delle modifiche può richiedere alcuni minuti, ma dopo aver eseguito questo comando, le richieste che contengono un corpo di dimensioni superiori a 1024 byte verranno rifiutate da. AWS AppSync

Note

Dopo aver creato un nuovo ACL Web nella AWS WAF Console, possono essere necessari alcuni minuti prima che l'ACL Web sia disponibile per l'associazione a un'API. Se esegui il comando CLI e ricevi un `WAFUnavailableEntityException` errore, attendi qualche minuto e riprova a eseguire il comando.

Example regola web ACL per limitare le richieste provenienti da un singolo indirizzo IP

Di seguito è riportato un esempio di regola che limita un' AppSync API a 100 richieste da un singolo indirizzo IP. Questo verrebbe inserito nell'editor Rule JSON durante la creazione di un ACL Web con una regola basata sulla velocità nella Console. AWS WAF

```
{
  "Name": "Throttle",
  "Priority": 0,
  "Action": {
    "Block": {}
  },
  "VisibilityConfig": {
    "SampledRequestsEnabled": true,
    "CloudWatchMetricsEnabled": true,
    "MetricName": "Throttle"
  },
  "Statement": {
    "RateBasedStatement": {
      "Limit": 100,
      "AggregateKeyType": "IP"
    }
  }
}
```

```
}
}
```

Dopo aver creato l'ACL Web utilizzando la regola di esempio precedente, è necessario associarlo alla propria API. AppSync. È possibile eseguire questo passaggio AWS CLI eseguendo il comando seguente.

```
aws waf associate-web-acl --web-acl-id waf-web-acl-arn --resource-arn appsync-api-arn
```

Example regola web ACL per impedire le query di introspezione GraphQL `__schema` su un'API

Di seguito è riportato un esempio di regola che impedisce le query di introspezione GraphQL `__schema` su un'API. Qualsiasi corpo HTTP che include la stringa «`__schema`» verrà bloccato. Questo verrebbe inserito nell'editor Rule JSON durante la creazione di un ACL web nella console. AWS WAF

```
{
  "Name": "BodyRule",
  "Priority": 5,
  "Action": {
    "Block": {}
  },
  "VisibilityConfig": {
    "SampledRequestsEnabled": true,
    "CloudWatchMetricsEnabled": true,
    "MetricName": "BodyRule"
  },
  "Statement": {
    "ByteMatchStatement": {
      "FieldToMatch": {
        "Body": {}
      },
      "PositionalConstraint": "CONTAINS",
      "SearchString": "__schema",
      "TextTransformations": [
        {
          "Type": "NONE",
          "Priority": 0
        }
      ]
    }
  }
}
```

```
}
```

Dopo aver creato l'ACL Web utilizzando la regola di esempio precedente, è necessario associarlo all'API. AppSync. È possibile eseguire questo passaggio AWS CLI eseguendo il comando seguente.

```
aws waf associate-web-acl --web-acl-id waf-web-acl-arn --resource-arn appsync-api-arn
```

Sicurezza in AWS AppSync

La sicurezza del cloud AWS è la massima priorità. In qualità di AWS cliente, puoi beneficiare di data center e architetture di rete progettati per soddisfare i requisiti delle organizzazioni più sensibili alla sicurezza.

La sicurezza è una responsabilità condivisa tra te e te. AWS Il [modello di responsabilità condivisa](#) descrive questo aspetto come sicurezza del cloud e sicurezza nel cloud:

- **Sicurezza del cloud:** AWS è responsabile della protezione dell'infrastruttura che gestisce AWS i servizi nel AWS cloud. AWS ti fornisce anche servizi che puoi utilizzare in modo sicuro. I revisori esterni testano e verificano regolarmente l'efficacia della nostra sicurezza nell'ambito dei [AWS Programmi di AWS conformità dei Programmi di conformità](#) dei di . Per maggiori informazioni sui programmi di conformità applicabili AWS AppSync, consulta la sezione [AWS Servizi rientranti nell'ambito del programma di conformitàAWS](#) .
- **Sicurezza nel cloud:** la tua responsabilità è determinata dal AWS servizio che utilizzi. L'utente è anche responsabile di altri fattori, tra cui la riservatezza dei dati, i requisiti della propria azienda e le leggi e normative vigenti.

Questa documentazione ti aiuta a capire come applicare il modello di responsabilità condivisa durante l'utilizzo AWS AppSync. Negli argomenti seguenti viene illustrato come eseguire la configurazione AWS AppSync per soddisfare gli obiettivi di sicurezza e conformità. Imparerai anche a utilizzare altri AWS servizi che ti aiutano a monitorare e proteggere AWS AppSync le tue risorse.

Argomenti

- [Protezione dei dati in AWS AppSync](#)
- [Convalida della conformità per AWS AppSync](#)
- [Sicurezza dell'infrastruttura in AWS AppSync](#)
- [Resilienza in AWS AppSync](#)
- [Gestione delle identità e degli accessi per AWS AppSync](#)
- [Registrazione delle chiamate AWS AppSync API con AWS CloudTrail](#)
- [Best practice di sicurezza per AWS AppSync](#)

Protezione dei dati in AWS AppSync

Il modello di [responsabilità AWS condivisa modello](#) di si applica alla protezione dei dati in AWS AppSync. Come descritto in questo modello, AWS è responsabile della protezione dell'infrastruttura globale che gestisce tutti i Cloud AWS. L'utente è responsabile del controllo dei contenuti ospitati su questa infrastruttura. L'utente è inoltre responsabile della configurazione della protezione e delle attività di gestione per i Servizi AWS utilizzati. Per maggiori informazioni sulla privacy dei dati, consulta le [Domande frequenti sulla privacy dei dati](#). Per informazioni sulla protezione dei dati in Europa, consulta il post del blog relativo al [AWS Modello di responsabilità condivisa e GDPR](#) nel AWS Blog sulla sicurezza.

Ai fini della protezione dei dati, consigliamo di proteggere Account AWS le credenziali e configurare i singoli utenti con AWS IAM Identity Center or AWS Identity and Access Management (IAM). In tal modo, a ogni utente verranno assegnate solo le autorizzazioni necessarie per svolgere i suoi compiti. Suggeriamo, inoltre, di proteggere i dati nei seguenti modi:

- Utilizza l'autenticazione a più fattori (MFA) con ogni account.
- SSL/TLS Da utilizzare per comunicare con AWS le risorse. È richiesto TLS 1.2 ed è consigliato TLS 1.3.
- Configura l'API e la registrazione delle attività degli utenti con AWS CloudTrail. Per informazioni sull'utilizzo dei CloudTrail percorsi per acquisire AWS le attività, consulta [Lavorare con i CloudTrail percorsi](#) nella Guida per l'AWS CloudTrail utente.
- Utilizza soluzioni di AWS crittografia, insieme a tutti i controlli di sicurezza predefiniti all'interno Servizi AWS.
- Utilizza i servizi di sicurezza gestiti avanzati, come Amazon Macie, che aiutano a individuare e proteggere i dati sensibili archiviati in Amazon S3.
- Se hai bisogno di moduli crittografici convalidati FIPS 140-3 per accedere AWS tramite un'interfaccia a riga di comando o un'API, usa un endpoint FIPS. Per ulteriori informazioni sugli endpoint FIPS disponibili, consulta il [Federal Information Processing Standard \(FIPS\) 140-3](#).

Ti consigliamo di non inserire mai informazioni riservate o sensibili, ad esempio gli indirizzi e-mail dei clienti, nei tag o nei campi di testo in formato libero, ad esempio nel campo Nome. Ciò include quando lavori AWS AppSync o Servizi AWS utilizzi la console, l'API o. AWS CLI AWS SDKs I dati inseriti nei tag o nei campi di testo in formato libero utilizzati per i nomi possono essere utilizzati per i la fatturazione o i log di diagnostica. Quando si fornisce un URL a un server esterno, suggeriamo

vivamente di non includere informazioni sulle credenziali nell'URL per convalidare la richiesta al server.

Crittografia in movimento

AWS AppSync, come tutti i AWS servizi, utilizza TLS1 .2 e versioni successive per la comunicazione quando si utilizza la rete AWS pubblicata APIs e SDKs.

L'utilizzo AWS AppSync con altri AWS servizi come Amazon DynamoDB garantisce la crittografia in transito: AWS tutti i servizi utilizzano TLS 1.2 e versioni successive per comunicare tra loro, se non diversamente specificato. Per i resolver che utilizzano Amazon EC2 CloudFront o, è tua responsabilità verificare che TLS (HTTPS) sia configurato e sicuro. Per informazioni sulla configurazione di HTTPS in Amazon EC2, [consulta SSL/TLS Configuring on Amazon Linux 2](#) nella guida per l'utente di Amazon EC2. Per informazioni sulla configurazione di HTTPS su CloudFront, consulta [HTTPS in Amazon CloudFront nella](#) guida per l' CloudFront utente.

Convalida della conformità per AWS AppSync

I revisori esterni valutano la sicurezza e la conformità nell' AWS AppSync ambito di più programmi di AWS conformità. AWS AppSync è conforme ai programmi SOC, PCI, HIPAA/HIPAA BAA, IRAP, C5, ENS High, OSPAR e HITRUST CSF.

Per sapere se un Servizio AWS programma rientra nell'ambito di specifici programmi di conformità, consulta Scope by Compliance Program [Servizi AWS in Scope by Compliance Program](#) e che ti interessa. Per informazioni generali, consulta Programmi di [AWS conformità Programmi](#) di di .

È possibile scaricare report di audit di terze parti utilizzando AWS Artifact. Per ulteriori informazioni, consulta [Scaricamento dei report in AWS Artifact](#) .

La vostra responsabilità di conformità durante l'utilizzo Servizi AWS è determinata dalla sensibilità dei dati, dagli obiettivi di conformità dell'azienda e dalle leggi e dai regolamenti applicabili. Per ulteriori informazioni sulla responsabilità di conformità durante l'utilizzo Servizi AWS, consulta la [Documentazione AWS sulla sicurezza](#).

Sicurezza dell'infrastruttura in AWS AppSync

In quanto servizio gestito, AWS AppSync è protetto dalla sicurezza di rete AWS globale. Per informazioni sui servizi AWS di sicurezza e su come AWS protegge l'infrastruttura, consulta [AWS](#)

[Cloud Security](#). Per progettare il tuo AWS ambiente utilizzando le migliori pratiche per la sicurezza dell'infrastruttura, vedi [Infrastructure Protection](#) in Security Pillar AWS Well-Architected Framework.

Utilizzate chiamate API AWS pubblicate per accedere AWS AppSync attraverso la rete. I client devono supportare quanto segue:

- Transport Layer Security (TLS). È richiesto TLS 1.2 ed è consigliato TLS 1.3.
- Suite di cifratura con Perfect Forward Secrecy (PFS), ad esempio Ephemeral Diffie-Hellman (DHE) o Elliptic Curve Ephemeral Diffie-Hellman (ECDHE). La maggior parte dei sistemi moderni, come Java 7 e versioni successive, supporta tali modalità.

Resilienza in AWS AppSync

L'infrastruttura AWS globale è costruita attorno a AWS regioni e zone di disponibilità. AWS Le regioni forniscono più zone di disponibilità fisicamente separate e isolate, collegate con reti a bassa latenza, ad alto throughput e altamente ridondanti. Con le zone di disponibilità è possibile progettare e gestire applicazioni e database che eseguono automaticamente il failover tra zone di disponibilità senza interruzioni. Le zone di disponibilità sono più disponibili, tolleranti ai guasti e scalabili rispetto alle infrastrutture a data center singolo o multiplo tradizionali.

[Per ulteriori informazioni su AWS regioni e zone di disponibilità, consulta Global Infrastructure.AWS](#)

[Oltre all'infrastruttura AWS globale, AWS AppSync consente di definire la maggior parte delle risorse utilizzando AWS CloudFormation modelli; per un esempio di utilizzo dei modelli per dichiarare AWS AppSync le risorse, consulta Practical use cases for AWS AppSync Pipeline Resolvers sul AWS blog e nella Guida per l'utente. CloudFormationAWS CloudFormation](#)

Gestione delle identità e degli accessi per AWS AppSync

AWS Identity and Access Management (IAM) è un software Servizio AWS che aiuta un amministratore a controllare in modo sicuro l'accesso alle AWS risorse. Gli amministratori IAM controllano chi può essere autenticato (effettuato l'accesso) e autorizzato (disporre delle autorizzazioni) a utilizzare le risorse. AWS AppSync IAM è uno Servizio AWS strumento che puoi utilizzare senza costi aggiuntivi.

Argomenti

- [Destinatari](#)

- [Autenticazione con identità](#)
- [Gestione dell'accesso tramite policy](#)
- [Come AWS AppSync funziona con IAM](#)
- [Politiche basate sull'identità per AWS AppSync](#)
- [Risoluzione dei problemi di AWS AppSync identità e accesso](#)

Destinatari

Il modo in cui utilizzi AWS Identity and Access Management (IAM) varia in base al tuo ruolo:

- Utente del servizio: richiedi le autorizzazioni all'amministratore se non riesci ad accedere alle funzionalità (consulta [Risoluzione dei problemi di AWS AppSync identità e accesso](#))
- Amministratore del servizio: determina l'accesso degli utenti e invia le richieste di autorizzazione (consulta [Come AWS AppSync funziona con IAM](#))
- Amministratore IAM: scrivi policy per gestire l'accesso (consulta [Politiche basate sull'identità per AWS AppSync](#))

Autenticazione con identità

L'autenticazione è il modo in cui accedi AWS utilizzando le tue credenziali di identità. Devi autenticarti come utente IAM o assumendo un ruolo IAM. Utente root dell'account AWS

Puoi accedere come identità federata utilizzando credenziali provenienti da una fonte di identità come AWS IAM Identity Center (IAM Identity Center), autenticazione Single Sign-On o credenziali. Google/Facebook Per ulteriori informazioni sull'accesso, consulta [Come accedere all' Account AWS](#) nella Guida per l'utente di Accedi ad AWS .

Per l'accesso programmatico, AWS fornisce un SDK e una CLI per firmare crittograficamente le richieste. Per ulteriori informazioni, consulta [AWS Signature Version 4 per le richieste API](#) nella Guida per l'utente di IAM.

Account AWS utente root

Quando si crea un Account AWS, si inizia con un'identità di accesso denominata utente Account AWS root che ha accesso completo a tutte Servizi AWS le risorse. Consigliamo vivamente di non utilizzare l'utente root per le attività quotidiane. Per le attività che richiedono le credenziali dell'utente root, consulta [Attività che richiedono le credenziali dell'utente root](#) nella Guida per l'utente IAM.

Identità federata

Come procedura ottimale, richiedi agli utenti umani di utilizzare la federazione con un provider di identità per accedere Servizi AWS utilizzando credenziali temporanee.

Un'identità federata è un utente della directory aziendale, del provider di identità Web o Directory Service che accede Servizi AWS utilizzando le credenziali di una fonte di identità. Le identità federate assumono ruoli che forniscono credenziali temporanee.

Per la gestione centralizzata degli accessi, si consiglia di utilizzare AWS IAM Identity Center. Per ulteriori informazioni, consulta [Che cos'è il Centro identità IAM?](#) nella Guida per l'utente di AWS IAM Identity Center .

Utenti e gruppi IAM

Un [utente IAM](#) è una identità che dispone di autorizzazioni specifiche per una singola persona o applicazione. Ti consigliamo di utilizzare credenziali temporanee invece di utenti IAM con credenziali a lungo termine. Per ulteriori informazioni, consulta [Richiedere agli utenti umani di utilizzare la federazione con un provider di identità per accedere AWS utilizzando credenziali temporanee](#) nella Guida per l'utente IAM.

Un [gruppo IAM](#) specifica una raccolta di utenti IAM e semplifica la gestione delle autorizzazioni per gestire gruppi di utenti di grandi dimensioni. Per ulteriori informazioni, consulta [Casi d'uso per utenti IAM](#) nella Guida per l'utente di IAM.

Ruoli IAM

Un [ruolo IAM](#) è un'identità con autorizzazioni specifiche che fornisce credenziali temporanee. Puoi assumere un ruolo [passando da un ruolo utente a un ruolo IAM \(console\)](#) o chiamando un'operazione AWS CLI o AWS API. Per ulteriori informazioni, consulta [Metodi per assumere un ruolo](#) nella Guida per l'utente di IAM.

I ruoli IAM sono utili per l'accesso degli utenti federati, le autorizzazioni utente IAM temporanee, l'accesso multi-account, l'accesso multi-servizio e le applicazioni in esecuzione su Amazon EC2. Per maggiori informazioni, consultare [Accesso a risorse multi-account in IAM](#) nella Guida per l'utente IAM.

Gestione dell'accesso tramite policy

Puoi controllare l'accesso AWS creando policy e associandole a AWS identità o risorse. Una policy definisce le autorizzazioni quando è associata a un'identità o a una risorsa. AWS valuta queste

politiche quando un preside effettua una richiesta. La maggior parte delle politiche viene archiviata AWS come documenti JSON. Per maggiori informazioni sui documenti delle policy JSON, consulta [Panoramica delle policy JSON](#) nella Guida per l'utente IAM.

Utilizzando le policy, gli amministratori specificano chi ha accesso a cosa definendo quale principale può eseguire azioni su quali risorse e in quali condizioni.

Per impostazione predefinita, utenti e ruoli non dispongono di autorizzazioni. Un amministratore IAM crea le policy IAM e le aggiunge ai ruoli, che gli utenti possono quindi assumere. Le policy IAM definiscono le autorizzazioni indipendentemente dal metodo utilizzato per eseguirle.

Policy basate sull'identità

Le policy basate su identità sono documenti di policy di autorizzazione JSON che è possibile collegare a un'identità (utente, gruppo o ruolo). Tali policy controllano le operazioni autorizzate per l'identità, nonché le risorse e le condizioni in cui possono essere eseguite. Per informazioni su come creare una policy basata su identità, consultare [Definizione di autorizzazioni personalizzate IAM con policy gestite dal cliente](#) nella Guida per l'utente IAM.

Le policy basate su identità possono essere policy in linea (con embedding direttamente in una singola identità) o policy gestite (policy autonome collegate a più identità). Per informazioni su come scegliere tra una policy gestita o una policy inline, consulta [Scegliere tra policy gestite e policy in linea](#) nella Guida per l'utente di IAM.

Policy basate sulle risorse

Le policy basate su risorse sono documenti di policy JSON che è possibile collegare a una risorsa. Gli esempi includono le policy di trust dei ruoli IAM e le policy dei bucket di Amazon S3. Nei servizi che supportano policy basate sulle risorse, gli amministratori dei servizi possono utilizzarli per controllare l'accesso a una risorsa specifica. In una policy basata sulle risorse è obbligatorio [specificare un'entità principale](#).

Le policy basate sulle risorse sono policy inline che si trovano in tale servizio. Non è possibile utilizzare le policy AWS gestite di IAM in una policy basata sulle risorse.

Altri tipi di policy

AWS supporta tipi di policy aggiuntivi che possono impostare le autorizzazioni massime concesse dai tipi di policy più comuni:

- Limiti delle autorizzazioni: imposta il numero massimo di autorizzazioni che una policy basata su identità ha la possibilità di concedere a un'entità IAM. Per ulteriori informazioni, consulta [Limiti delle autorizzazioni per le entità IAM](#) nella Guida per l'utente di IAM.
- Politiche di controllo del servizio (SCPs): specificano le autorizzazioni massime per un'organizzazione o un'unità organizzativa in AWS Organizations. Per ulteriori informazioni, consultare [Policy di controllo dei servizi](#) nella Guida per l'utente di AWS Organizations.
- Politiche di controllo delle risorse (RCPs): imposta le autorizzazioni massime disponibili per le risorse nei tuoi account. Per ulteriori informazioni, consulta [Politiche di controllo delle risorse \(RCPs\)](#) nella Guida per l'AWS Organizations utente.
- Policy di sessione: policy avanzate passate come parametro quando si crea una sessione temporanea per un ruolo o un utente federato. Per maggiori informazioni, consultare [Policy di sessione](#) nella Guida per l'utente IAM.

Più tipi di policy

Quando a una richiesta si applicano più tipi di policy, le autorizzazioni risultanti sono più complicate da comprendere. Per scoprire come si AWS determina se consentire o meno una richiesta quando sono coinvolti più tipi di policy, consulta [Logica di valutazione delle policy](#) nella IAM User Guide.

Come AWS AppSync funziona con IAM

Prima di utilizzare IAM per gestire l'accesso a AWS AppSync, scopri con quali funzionalità IAM è disponibile l'uso AWS AppSync.

Funzionalità IAM che puoi utilizzare con AWS AppSync

Funzionalità IAM	AWS AppSync supporto
Policy basate sull'identità	Sì
Policy basate su risorse	No
Operazioni di policy	Sì
Risorse relative alle policy	Sì
Chiavi di condizione delle policy	No

Funzionalità IAM	AWS AppSync supporto
ACLs	No
ABAC (tag nelle policy)	Parziale
Credenziali temporanee	Sì
Inoltro delle sessioni di accesso (FAS)	Parziale
Ruoli di servizio	No
Ruoli collegati al servizio	Parziale

Per avere una panoramica di alto livello su come AWS AppSync e altri AWS servizi funzionano con la maggior parte delle funzionalità IAM, consulta [AWS i servizi che funzionano con IAM nella IAM](#) User Guide.

Politiche basate sull'identità per AWS AppSync

Supporta le policy basate sull'identità: sì

Le policy basate sull'identità sono documenti di policy di autorizzazione JSON che è possibile allegare a un'identità (utente, gruppo di utenti o ruolo IAM). Tali policy definiscono le operazioni che utenti e ruoli possono eseguire, su quali risorse e in quali condizioni. Per informazioni su come creare una policy basata su identità, consulta [Definizione di autorizzazioni personalizzate IAM con policy gestite dal cliente](#) nella Guida per l'utente di IAM.

Con le policy basate sull'identità di IAM, è possibile specificare quali operazioni e risorse sono consentite o respinte, nonché le condizioni in base alle quali le operazioni sono consentite o respinte. Per informazioni su tutti gli elementi utilizzabili in una policy JSON, consulta [Guida di riferimento agli elementi delle policy JSON IAM](#) nella Guida per l'utente IAM.

Esempi di politiche basate sull'identità per AWS AppSync

Per visualizzare esempi di politiche basate sull' AWS AppSync identità, vedere. [Politiche basate sull'identità per AWS AppSync](#)

Politiche basate sulle risorse all'interno AWS AppSync

Supporta le policy basate su risorse: no

Le policy basate su risorse sono documenti di policy JSON che è possibile collegare a una risorsa. Esempi di policy basate sulle risorse sono le policy di attendibilità dei ruoli IAM e le policy di bucket Amazon S3. Nei servizi che supportano policy basate sulle risorse, gli amministratori dei servizi possono utilizzarli per controllare l'accesso a una risorsa specifica. Quando è collegata a una risorsa, una policy definisce le operazioni che un principale può eseguire su tale risorsa e a quali condizioni. In una policy basata sulle risorse è obbligatorio [specificare un'entità principale](#). I principali possono includere account, utenti, ruoli, utenti federati o. Servizi AWS

Per consentire l'accesso multi-account, è possibile specificare un intero account o entità IAM in un altro account come entità principale in una policy basata sulle risorse. Per ulteriori informazioni, consulta [Accesso a risorse multi-account in IAM](#) nella Guida per l'utente IAM.

Azioni politiche per AWS AppSync

Supporta le operazioni di policy: si

Gli amministratori possono utilizzare le policy AWS JSON per specificare chi ha accesso a cosa. In altre parole, quale entità principale può eseguire operazioni su quali risorse e in quali condizioni.

L'elemento `Action` di una policy JSON descrive le operazioni che è possibile utilizzare per consentire o negare l'accesso in una policy. Includere le operazioni in una policy per concedere le autorizzazioni a eseguire l'operazione associata.

Per visualizzare un elenco di AWS AppSync azioni, vedere [Azioni definite da AWS AppSync](#) nel Service Authorization Reference.

Le azioni politiche in AWS AppSync uso utilizzano il seguente prefisso prima dell'azione:

```
appsync
```

Per specificare più operazioni in una sola istruzione, occorre separarle con la virgola.

```
"Action": [  
  "appsync:action1",  
  "appsync:action2"
```

```
]
```

Per visualizzare esempi di politiche AWS AppSync basate sull'identità, vedere. [Politiche basate sull'identità per AWS AppSync](#)

Risorse politiche per AWS AppSync

Supporta le risorse relative alle policy: sì

Gli amministratori possono utilizzare le policy AWS JSON per specificare chi ha accesso a cosa. In altre parole, quale entità principale può eseguire operazioni su quali risorse e in quali condizioni.

L'elemento JSON `Resource` della policy specifica l'oggetto o gli oggetti ai quali si applica l'operazione. Come best practice, specifica una risorsa utilizzando il suo [nome della risorsa Amazon \(ARN\)](#). Per le azioni che non supportano le autorizzazioni a livello di risorsa, si utilizza un carattere jolly (*) per indicare che l'istruzione si applica a tutte le risorse.

```
"Resource": "*" 
```

Per visualizzare un elenco dei tipi di AWS AppSync risorse e relativi ARNs, vedere [Resources defined by AWS AppSync](#) nel Service Authorization Reference. Per informazioni sulle operazioni con cui è possibile specificare l'ARN di ogni risorsa, consulta la sezione [Operazioni definite da AWS AppSync](#).

Per visualizzare esempi di politiche AWS AppSync basate sull'identità, vedere. [Politiche basate sull'identità per AWS AppSync](#)

Chiavi relative alle condizioni delle politiche per AWS AppSync

Supporta le chiavi di condizione delle policy specifiche del servizio: No

Gli amministratori possono utilizzare le policy AWS JSON per specificare chi ha accesso a cosa. In altre parole, quale entità principale può eseguire operazioni su quali risorse e in quali condizioni.

L'elemento `Condition` specifica quando le istruzioni vengono eseguite in base a criteri definiti. È possibile compilare espressioni condizionali che utilizzano [operatori di condizione](#), ad esempio

uguale a o minore di, per soddisfare la condizione nella policy con i valori nella richiesta. Per visualizzare tutte le chiavi di condizione AWS globali, consulta le chiavi di [contesto delle condizioni AWS globali nella Guida](#) per l'utente IAM.

Per visualizzare un elenco di chiavi di AWS AppSync condizione, consulta [Condition keys for AWS AppSync](#) nel Service Authorization Reference. Per sapere con quali azioni e risorse puoi utilizzare una chiave di condizione, vedi [Azioni definite da AWS AppSync](#).

Per visualizzare esempi di politiche AWS AppSync basate sull'identità, vedere. [Politiche basate sull'identità per AWS AppSync](#)

Liste di controllo degli accessi (ACL) in ACLs AWS AppSync

Supporti ACLs: no

Le liste di controllo degli accessi (ACLs) controllano quali principali (membri dell'account, utenti o ruoli) dispongono delle autorizzazioni per accedere a una risorsa. ACLs sono simili alle politiche basate sulle risorse, sebbene non utilizzino il formato del documento di policy JSON.

Controllo degli accessi basato sugli attributi (ABAC) con AWS AppSync

Supporta ABAC (tag nelle policy): parzialmente

Il controllo degli accessi basato su attributi (ABAC) è una strategia di autorizzazione che definisce le autorizzazioni in base agli attributi, chiamati tag. Puoi allegare tag a entità e AWS risorse IAM, quindi progettare politiche ABAC per consentire le operazioni quando il tag del principale corrisponde al tag sulla risorsa.

Per controllare l'accesso basato su tag, fornire informazioni sui tag nell'[elemento condizione](#) di una policy utilizzando le chiavi di condizione `aws:ResourceTag/key-name`, `aws:RequestTag/key-name` o `aws:TagKeys`.

Se un servizio supporta tutte e tre le chiavi di condizione per ogni tipo di risorsa, il valore per il servizio è Sì. Se un servizio supporta tutte e tre le chiavi di condizione solo per alcuni tipi di risorsa, allora il valore sarà Parziale.

Per maggiori informazioni su ABAC, consulta [Definizione delle autorizzazioni con autorizzazione ABAC](#) nella Guida per l'utente di IAM. Per visualizzare un tutorial con i passaggi per l'impostazione di ABAC, consulta [Utilizzo del controllo degli accessi basato su attributi \(ABAC\)](#) nella Guida per l'utente di IAM.

Utilizzo di credenziali temporanee con AWS AppSync

Supporta le credenziali temporanee: sì

Le credenziali temporanee forniscono l'accesso a breve termine alle AWS risorse e vengono create automaticamente quando si utilizza la federazione o si cambia ruolo. AWS consiglia di generare dinamicamente credenziali temporanee anziché utilizzare chiavi di accesso a lungo termine. Per ulteriori informazioni, consulta [Credenziali di sicurezza temporanee in IAM](#) e [Servizi AWS compatibili con IAM](#) nella Guida per l'utente IAM.

Sessioni di accesso diretto per AWS AppSync

Supporta sessioni di accesso diretto (FAS): parziali

Le sessioni di accesso inoltrato (FAS) utilizzano le autorizzazioni del principale chiamante e Servizio AWS, in combinazione con la richiesta, Servizio AWS per effettuare richieste ai servizi downstream. Per i dettagli delle policy relative alle richieste FAS, consulta [Forward access sessions](#).

Ruoli di servizio per AWS AppSync

Supporta i ruoli di servizio: no

Un ruolo di servizio è un [ruolo IAM](#) che un servizio assume per eseguire operazioni per tuo conto. Un amministratore IAM può creare, modificare ed eliminare un ruolo di servizio dall'interno di IAM. Per ulteriori informazioni, consulta [Create a role to delegate permissions to an Servizio AWS](#) nella Guida per l'utente IAM.

Warning

La modifica delle autorizzazioni per un ruolo di servizio potrebbe compromettere AWS AppSync la funzionalità. Modifica i ruoli di servizio solo quando viene AWS AppSync fornita una guida in tal senso.

Ruoli collegati ai servizi per AWS AppSync

Supporta i ruoli legati ai servizi: Parziale

Un ruolo collegato al servizio è un tipo di ruolo di servizio collegato a un. Servizio AWS Il servizio può assumere il ruolo per eseguire un'operazione per tuo conto. I ruoli collegati al servizio vengono

visualizzati nel tuo account Account AWS e sono di proprietà del servizio. Un amministratore IAM può visualizzare le autorizzazioni per i ruoli collegati al servizio, ma non modificarle.

Per i dettagli sulla creazione o la gestione di ruoli collegati ai servizi, consulta i [AWS servizi che funzionano con IAM nella IAM](#) User Guide. Trova un servizio nella tabella che include un Yes nella colonna Service-linked role (Ruolo collegato ai servizi). Scegli il collegamento Sì per visualizzare la documentazione relativa al ruolo collegato ai servizi per tale servizio.

Politiche basate sull'identità per AWS AppSync

Per impostazione predefinita, gli utenti e i ruoli non dispongono dell'autorizzazione per creare o modificare risorse AWS AppSync. Per concedere agli utenti l'autorizzazione a eseguire azioni sulle risorse di cui hanno bisogno, un amministratore IAM può creare policy IAM.

Per informazioni su come creare una policy basata su identità IAM utilizzando questi documenti di policy JSON di esempio, consulta [Creazione di policy IAM \(console\)](#) nella Guida per l'utente di IAM.

Per informazioni dettagliate sulle azioni e sui tipi di risorse definiti da AWS AppSync, incluso il formato di ARNs per ogni tipo di risorsa, vedere [Azioni, risorse e chiavi di condizione AWS AppSync nel Service Authorization](#) Reference.

Per conoscere le best practice per la creazione e la configurazione di policy basate sull'identità IAM, consulta. [the section called “Le migliori pratiche in materia di policy IAM”](#)

Per un elenco delle politiche basate sull'identità IAM per, consulta. AWS AppSync [AWS politiche gestite per AWS AppSync](#)

Argomenti

- [Utilizzo della console AWS AppSync](#)
- [Consentire agli utenti di visualizzare le loro autorizzazioni](#)
- [Accesso a un bucket Amazon S3](#)
- [Visualizzazione in base ai tag AWS AppSync widgets](#)
- [AWS politiche gestite per AWS AppSync](#)

Utilizzo della console AWS AppSync

Per accedere alla AWS AppSync console, è necessario disporre di un set minimo di autorizzazioni. Queste autorizzazioni devono consentirti di elencare e visualizzare i dettagli sulle AWS AppSync risorse del tuo. Account AWS Se crei una policy basata sull'identità più restrittiva rispetto alle

autorizzazioni minime richieste, la console non funzionerà nel modo previsto per le entità (utenti o ruoli) associate a tale policy.

Non è necessario consentire autorizzazioni minime per la console per gli utenti che effettuano chiamate solo verso AWS CLI o l' AWS API. Al contrario, è opportuno concedere l'accesso solo alle azioni che corrispondono all'operazione API che stanno cercando di eseguire.

Per garantire che gli utenti e i ruoli IAM possano continuare a utilizzare la AWS AppSync console, collega anche la policy AWS AppSync ConsoleAccess o la policy ReadOnly AWS gestita alle entità. Per maggiori informazioni, consulta [Aggiunta di autorizzazioni a un utente](#) nella Guida per l'utente di IAM.

Consentire agli utenti di visualizzare le loro autorizzazioni

Questo esempio mostra in che modo è possibile creare una policy che consente agli utenti IAM di visualizzare le policy inline e gestite che sono collegate alla relativa identità utente. Questa policy include le autorizzazioni per completare questa azione sulla console o utilizzando programmaticamente l' AWS CLI API o. AWS

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "ViewOwnUserInfo",
      "Effect": "Allow",
      "Action": [
        "iam:GetUserPolicy",
        "iam:ListGroupsForUser",
        "iam:ListAttachedUserPolicies",
        "iam:ListUserPolicies",
        "iam:GetUser"
      ],
      "Resource": ["arn:aws:iam::*:user/${aws:username}"]
    },
    {
      "Sid": "NavigateInConsole",
      "Effect": "Allow",
      "Action": [
        "iam:GetGroupPolicy",
        "iam:GetPolicyVersion",
        "iam:GetPolicy",
        "iam:ListAttachedGroupPolicies",
```

```

        "iam:ListGroupPolicies",
        "iam:ListPolicyVersions",
        "iam:ListPolicies",
        "iam:ListUsers"
    ],
    "Resource": "*"
}
]
}

```

Accesso a un bucket Amazon S3

In questo esempio, vuoi concedere a un utente IAM del tuo AWS account l'accesso a uno dei tuoi bucket Amazon S3, `examplebucket`. Si vuole anche consentire all'utente di aggiungere, aggiornare ed eliminare oggetti.

Oltre ad assegnare le autorizzazioni `s3:PutObject`, `s3:GetObject` e `s3:DeleteObject` all'utente, la policy assegna anche le autorizzazioni `s3:ListAllMyBuckets`, `s3:GetBucketLocation` e `s3:ListBucket`. Queste sono le autorizzazioni aggiuntive richieste dalla console. Inoltre, le operazioni `s3:PutObjectAcl` e `s3:GetObjectAcl` sono necessarie per essere in grado di copiare, tagliare e incollare gli oggetti nella console. Per un esempio di procedura dettagliata che concede le autorizzazioni agli utenti e li verifica utilizzando la console, consulta [Un esempio di procedura dettagliata: Using user policy to control access to your bucket.](#)

JSON

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "ListBucketsInConsole",
      "Effect": "Allow",
      "Action": [
        "s3:ListAllMyBuckets"
      ],
      "Resource": "arn:aws:s3:::*"
    },
    {
      "Sid": "ViewSpecificBucketInfo",
      "Effect": "Allow",
      "Action": [

```

```

        "s3:ListBucket",
        "s3:GetBucketLocation"
    ],
    "Resource": "arn:aws:s3:::examplebucket"
},
{
    "Sid": "ManageBucketContents",
    "Effect": "Allow",
    "Action": [
        "s3:PutObject",
        "s3:PutObjectAcl",
        "s3:GetObject",
        "s3:GetObjectAcl",
        "s3:DeleteObject"
    ],
    "Resource": "arn:aws:s3:::examplebucket/*"
}
]
}

```

Visualizzazione in base ai tag AWS AppSync *widgets*

È possibile utilizzare le condizioni nella policy basata sulle identità per controllare l'accesso alle risorse di AWS AppSync in base ai tag. Questo esempio mostra come creare una policy che consente di visualizzare una *widget*. Tuttavia, l'autorizzazione viene concessa solo se il valore del tag `Owner widget` è quello del nome utente. Questa policy concede anche le autorizzazioni necessarie per completare questa azione nella console.

Puoi allegare questa policy agli utenti IAM nel tuo account. Se un utente denominato `richard-roe` tenta di visualizzare un *widget* AWS AppSync, il *widget* deve essere contrassegnato `Owner=richard-roe` o `owner=richard-roe`. In caso contrario l'accesso è negato. La chiave di tag di condizione `Owner` corrisponde a `Owner` e `owner` perché i nomi delle chiavi di condizione non effettuano la distinzione tra maiuscole e minuscole. Per maggiori informazioni, consultare la sezione [Elementi delle policy JSON di IAM: condizione](#) nella Guida per l'utente di IAM.

AWS politiche gestite per AWS AppSync

Per aggiungere autorizzazioni a utenti, gruppi e ruoli, è più facile utilizzare le policy AWS gestite che scriverle autonomamente. La [creazione di policy gestite dai clienti IAM](#) che forniscono al tuo team

solo le autorizzazioni di cui ha bisogno richiede tempo e competenza. Per iniziare rapidamente, puoi utilizzare le nostre politiche AWS gestite. Queste policy coprono i casi d'uso comuni e sono disponibili nell'account Account AWS. Per ulteriori informazioni sulle policy AWS gestite, consulta le [policy AWS gestite](#) nella IAM User Guide.

AWS i servizi mantengono e aggiornano le politiche AWS gestite. Non è possibile modificare le autorizzazioni nelle politiche AWS gestite. I servizi aggiungono occasionalmente autorizzazioni aggiuntive a una policy AWS gestita per supportare nuove funzionalità. Questo tipo di aggiornamento interessa tutte le identità (utenti, gruppi e ruoli) a cui è collegata la policy. È più probabile che i servizi aggiornino una politica AWS gestita quando viene lanciata una nuova funzionalità o quando diventano disponibili nuove operazioni. I servizi non rimuovono le autorizzazioni da una policy AWS gestita, quindi gli aggiornamenti delle policy non comprometteranno le autorizzazioni esistenti.

Inoltre, AWS supporta politiche gestite per le funzioni lavorative che si estendono su più servizi. Ad esempio, la policy `ReadOnlyAccess` AWS gestita fornisce l'accesso in sola lettura a tutti i AWS servizi e le risorse. Quando un servizio lancia una nuova funzionalità, AWS aggiunge autorizzazioni di sola lettura per nuove operazioni e risorse. Per l'elenco e la descrizione delle policy di funzione dei processi, consultare la sezione [Policy gestite da AWS per funzioni di processi](#) nella Guida per l'utente di IAM.

AWS politica gestita: `AWSApp SyncInvokeFullAccess`

Utilizza la policy `AWSAppSyncInvokeFullAccess` AWS gestita per consentire agli amministratori di accedere al AWS AppSync servizio tramite la console o in modo indipendente.

È possibile allegare la policy `AWSAppSyncInvokeFullAccess` alle identità IAM.

Dettagli delle autorizzazioni

Questa policy include le seguenti autorizzazioni:

- **AWS AppSync**— Consente l'accesso amministrativo completo a tutte le risorse in AWS AppSync

JSON

```
{
```

```
"Version": "2012-10-17",
"Statement": [
  {
    "Effect": "Allow",
    "Action": [
      "appsync:GraphQL",
      "appsync:GetGraphqlApi",
      "appsync:ListGraphqlApis",
      "appsync:ListApiKeys"
    ],
    "Resource": "*"
  }
]
```

AWS politica gestita: AWSApp SyncSchemaAuthor

Utilizza la policy AWSAppSyncSchemaAuthor AWS gestita per consentire agli utenti IAM di accedere per creare, aggiornare e interrogare i propri schemi GraphQL. Per informazioni su cosa possono fare gli utenti con queste autorizzazioni, consulta. [Progettazione di GraphQL APIs con AWS AppSync](#)

È possibile allegare la policy AWSAppSyncSchemaAuthor alle identità IAM.

Dettagli delle autorizzazioni

Questa policy include le seguenti autorizzazioni:

- AWS AppSync— Consente le seguenti azioni:
 - Creazione di schemi GraphQL
 - Consentire la creazione, la modifica e l'eliminazione di tipi, resolver e funzioni GraphQL
 - Valutazione della logica dei modelli di richiesta e risposta
 - Valutazione del codice con un runtime e un contesto
 - Invio di query GraphQL a GraphQL APIs
 - Recupero dei dati GraphQL

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "appsync:GraphQL",
        "appsync:CreateResolver",
        "appsync:CreateType",
        "appsync>DeleteResolver",
        "appsync>DeleteType",
        "appsync:GetResolver",
        "appsync:GetType",
        "appsync:GetDataSource",
        "appsync:GetSchemaCreationStatus",
        "appsync:GetIntrospectionSchema",
        "appsync:GetGraphqlApi",
        "appsync:ListTypes",
        "appsync:ListApiKeys",
        "appsync:ListResolvers",
        "appsync:ListDataSources",
        "appsync:ListGraphqlApis",
        "appsync:StartSchemaCreation",
        "appsync:UpdateResolver",
        "appsync:UpdateType",
        "appsync:TagResource",
        "appsync:UntagResource",
        "appsync:ListTagsForResource",
        "appsync:CreateFunction",
        "appsync:UpdateFunction",
        "appsync:GetFunction",
        "appsync>DeleteFunction",
        "appsync:ListFunctions",
        "appsync:ListResolversByFunction",
        "appsync:EvaluateMappingTemplate",
        "appsync:EvaluateCode"
      ],
      "Resource": "*"
    }
  ]
}
```

```
}
```

AWS politica gestita: AWSApp SyncPushToCloudWatchLogs

AWS AppSync utilizza Amazon CloudWatch per monitorare le prestazioni della tua applicazione generando log che puoi utilizzare per risolvere i problemi e ottimizzare le tue richieste GraphQL. Per ulteriori informazioni, consulta [Utilizzo CloudWatch per monitorare e registrare i dati dell'API GraphQL](#).

Utilizza la policy AWSAppSyncPushToCloudWatchLogs AWS gestita per consentire di inviare i log AWS AppSync all'account di un utente IAM. CloudWatch

È possibile allegare la policy AWSAppSyncPushToCloudWatchLogs alle identità IAM.

Dettagli delle autorizzazioni

Questa policy include le seguenti autorizzazioni:

- **CloudWatch Logs**— Consente di AWS AppSync creare gruppi di log e flussi con nomi specifici. AWS AppSync invia gli eventi di registro al flusso di registro specificato.

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents"
      ],
      "Resource": "*"
    }
  ]
}
```

AWS politica gestita: AWSApp SyncAdministrator

Utilizza la policy `AWSAppSyncAdministrator` AWS gestita per consentire agli amministratori di accedere a tutto AWS AppSync tranne che alla AWS console.

Puoi collegarti `AWSAppSyncAdministrator` alle tue entità IAM. AWS AppSync associa inoltre questa policy a un ruolo di servizio che gli consente di eseguire azioni per tuo conto.

Dettagli delle autorizzazioni

Questa policy include le seguenti autorizzazioni:

- **AWS AppSync**— Consente l'accesso amministrativo completo a tutte le risorse in AWS AppSync
- **IAM**— Consente le seguenti azioni:
 - Creazione di ruoli collegati ai servizi AWS AppSync per consentire l'analisi delle risorse in altri servizi per conto dell'utente
 - Eliminazione di ruoli collegati ai servizi
 - Trasferimento di ruoli collegati ai servizi ad altri AWS servizi per assumerli in un secondo momento ed eseguire azioni per conto dell'utente

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "appsync:*"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "iam:PassRole"
      ],
      "Resource": "*"
    }
  ]
}
```

```

    ],
    "Resource": "*",
    "Condition": {
        "StringEquals": {
            "iam:PassedToService": [
                "appsync.amazonaws.com"
            ]
        }
    }
},
{
    "Effect": "Allow",
    "Action": "iam:CreateServiceLinkedRole",
    "Resource": "*",
    "Condition": {
        "StringEquals": {
            "iam:AWSServiceName": "appsync.amazonaws.com"
        }
    }
},
{
    "Effect": "Allow",
    "Action": [
        "iam:DeleteServiceLinkedRole",
        "iam:GetServiceLinkedRoleDeletionStatus"
    ],
    "Resource": "arn:aws:iam::*:role/aws-service-role/
appsync.amazonaws.com/AWSServiceRoleForAppSync*"
}
]
}

```

AWS politica gestita: AWSApp SyncServiceRolePolicy

Utilizza la politica AWSAppSyncServiceRolePolicy AWS gestita per consentire l'accesso ai AWS servizi e alle risorse che AWS AppSync utilizza o gestisce.

Non è possibile collegare AWSAppSyncServiceRolePolicy alle entità IAM. Questa policy è associata a un ruolo collegato al servizio che consente di eseguire azioni AWS AppSync per conto dell'utente. Per ulteriori informazioni, consulta [Ruoli collegati ai servizi per AWS AppSync](#).

Dettagli delle autorizzazioni

Questa policy include le seguenti autorizzazioni:

- X-Ray— AWS AppSync utilizza AWS X-Ray per raccogliere dati sulle richieste effettuate all'interno dell'applicazione. Per ulteriori informazioni, consulta [Utilizzo AWS X-Ray per tracciare le richieste in AWS AppSync](#).

Questa politica consente le seguenti azioni:

- Recupero delle regole di campionamento e dei relativi risultati
- Invio di dati di traccia al demone X-Ray

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "xray:PutTraceSegments",
        "xray:PutTelemetryRecords",
        "xray:GetSamplingTargets",
        "xray:GetSamplingRules",
        "xray:GetSamplingStatisticSummaries"
      ],
      "Resource": [
        "*"
      ]
    }
  ]
}
```

AWS AppSync aggiornamenti alle politiche gestite AWS

Visualizza i dettagli sugli aggiornamenti delle politiche AWS gestite AWS AppSync da quando questo servizio ha iniziato a tenere traccia di queste modifiche. Per ricevere avvisi automatici sulle modifiche a questa pagina, iscriviti al feed RSS nella pagina della cronologia dei AWS AppSync documenti.

Modifica	Descrizione	Data
AWSAppSyncSchemaAuthor - Aggiornamento a una policy esistente	È stata aggiunta un'azione <code>EvaluateCode</code> politica per consentire agli utenti di valutare il codice con un runtime e un contesto.	7 febbraio 2023
AWSAppSyncSchemaAuthor - Aggiornamento a una policy esistente	Sono state aggiunte azioni politiche per consentire le funzioni di elenco, acquisizione, creazione, aggiornamento ed eliminazione per un'API. È stata aggiunta un'azione <code>EvaluateMappingTemplate</code> politica per consentire agli utenti di valutare la logica del modello di mappatura del resolver di richieste e risposte. Sono state aggiunte azioni politiche per consentire l'etichettatura delle risorse.	25 agosto 2022
AWS AppSync ha iniziato a tenere traccia delle modifiche	AWS AppSync ha iniziato a tenere traccia delle modifiche per le sue politiche AWS gestite.	25 agosto 2022

Risoluzione dei problemi di AWS AppSync identità e accesso

Utilizza le seguenti informazioni per aiutarti a diagnosticare e risolvere i problemi più comuni che potresti riscontrare quando lavori con un AWS AppSync IAM.

Non sono autorizzato a eseguire alcuna azione in AWS AppSync

Se ti Console di gestione AWS dice che non sei autorizzato a eseguire un'azione, devi contattare l'amministratore per ricevere assistenza. L'amministratore è la persona da cui si sono ricevuti il nome utente e la password.

L'errore di esempio seguente si verifica quando l'utente IAM `mateojackson` tenta di utilizzare la console per visualizzare i dettagli su una `my-example-widget` risorsa fittizia, ma non dispone delle autorizzazioni fittizie `appsync:GetWidget`.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
  appsync:GetWidget on resource: my-example-widget
```

In questo caso, Mateo chiede al suo amministratore di aggiornare le policy per poter accedere alla risorsa `my-example-widget` mediante l'operazione `appsync:GetWidget`.

Non sono autorizzato a eseguire iam: PassRole

Se ricevi un errore che indica che non sei autorizzato a eseguire l'operazione `iam:PassRole`, le tue policy devono essere aggiornate per poter passare un ruolo a AWS AppSync.

Alcuni Servizi AWS consentono di passare un ruolo esistente a quel servizio invece di creare un nuovo ruolo di servizio o un ruolo collegato al servizio. Per eseguire questa operazione, è necessario disporre delle autorizzazioni per trasmettere il ruolo al servizio.

L'errore di esempio seguente si verifica quando un utente IAM denominato `marymajor` cerca di utilizzare la console per eseguire un'operazione in AWS AppSync. Tuttavia, l'operazione richiede che il servizio disponga delle autorizzazioni concesse da un ruolo di servizio. Mary non dispone delle autorizzazioni per trasmettere il ruolo al servizio.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
  iam:PassRole
```

In questo caso, le policy di Mary devono essere aggiornate per poter eseguire l'operazione `iam:PassRole`.

Se hai bisogno di aiuto, contatta il tuo AWS amministratore. L'amministratore è colui che ti ha fornito le credenziali di accesso.

Desidero visualizzare le mie chiavi di accesso

Dopo aver creato le chiavi di accesso utente IAM, è possibile visualizzare il proprio ID chiave di accesso in qualsiasi momento. Tuttavia, non è possibile visualizzare nuovamente la chiave di accesso segreta. Se perdi la chiave segreta, dovrai creare una nuova coppia di chiavi di accesso.

Le chiavi di accesso sono composte da due parti: un ID chiave di accesso (ad esempio AKIAIOSFODNN7EXAMPLE) e una chiave di accesso segreta (ad esempio, wJa1rXUtnFEMI/K7MDENG/bPxrFiCYEXAMPLEKEY). Come un nome utente e una password, è necessario utilizzare sia l'ID chiave di accesso sia la chiave di accesso segreta insieme per autenticare le richieste dell'utente. Gestisci le tue chiavi di accesso in modo sicuro mentre crei il nome utente e la password.

Important

Non fornire le chiavi di accesso a terze parti, neppure per aiutare a [trovare l'ID utente canonico](#). In questo modo, potresti concedere a qualcuno l'accesso permanente al tuo Account AWS.

Quando crei una coppia di chiavi di accesso, ti viene chiesto di salvare l'ID chiave di accesso e la chiave di accesso segreta in una posizione sicura. La chiave di accesso segreta è disponibile solo al momento della creazione. Se si perde la chiave di accesso segreta, è necessario aggiungere nuove chiavi di accesso all'utente IAM. È possibile avere massimo due chiavi di accesso. Se se ne hanno già due, è necessario eliminare una coppia di chiavi prima di crearne una nuova. Per visualizzare le istruzioni, consulta [Gestione delle chiavi di accesso](#) nella Guida per l'utente di IAM.

Sono un amministratore e voglio consentire ad altri di accedere AWS AppSync

Per consentire ad altri di accedere AWS AppSync, devi concedere l'autorizzazione alle persone o alle applicazioni che necessitano dell'accesso. Se si utilizza AWS IAM Identity Center per gestire persone e applicazioni, si assegnano set di autorizzazioni a utenti o gruppi per definirne il livello di accesso. I set di autorizzazioni creano e assegnano automaticamente le policy IAM ai ruoli IAM associati alla persona o all'applicazione. Per ulteriori informazioni, consulta [Set di autorizzazioni](#) nella Guida per l'AWS IAM Identity Center utente.

Se non utilizzi IAM Identity Center, devi creare entità IAM (utenti o ruoli) per le persone o le applicazioni che necessitano di accesso. Dovrai quindi collegare all'entità una policy che conceda le

autorizzazioni corrette in AWS AppSync. Dopo aver concesso le autorizzazioni, fornisci le credenziali all'utente o allo sviluppatore dell'applicazione. Utilizzeranno tali credenziali per accedere. AWS Per ulteriori informazioni sulla creazione di utenti, gruppi, policy e autorizzazioni IAM, consulta [IAM Identities](#) and [Policies and permissions in IAM nella IAM User Guide](#).

Voglio consentire a persone esterne al mio AWS account di accedere alle mie risorse AWS AppSync

È possibile creare un ruolo con il quale utenti in altri account o persone esterne all'organizzazione possono accedere alle tue risorse. È possibile specificare chi è attendibile per l'assunzione del ruolo. Per i servizi che supportano politiche basate sulle risorse o liste di controllo degli accessi (ACLs), puoi utilizzare tali politiche per concedere alle persone l'accesso alle tue risorse.

Per maggiori informazioni, consulta gli argomenti seguenti:

- Per sapere se AWS AppSync supporta queste funzionalità, consulta. [Come AWS AppSync funziona con IAM](#)
- Per scoprire come fornire l'accesso alle tue risorse attraverso Account AWS le risorse di tua proprietà, consulta [Fornire l'accesso a un utente IAM in un altro Account AWS di tua proprietà](#) nella IAM User Guide.
- Per scoprire come fornire l'accesso alle tue risorse a terze parti Account AWS, consulta [Fornire l'accesso a soggetti Account AWS di proprietà di terze parti](#) nella Guida per l'utente IAM.
- Per informazioni su come fornire l'accesso tramite la federazione delle identità, consulta [Fornire l'accesso a utenti autenticati esternamente \(federazione delle identità\)](#) nella Guida per l'utente IAM.
- Per informazioni sulle differenze di utilizzo tra ruoli e policy basate su risorse per l'accesso multi-account, consulta [Accesso a risorse multi-account in IAM](#) nella Guida per l'utente di IAM.

Registrazione delle chiamate AWS AppSync API con AWS CloudTrail

AWS AppSync è integrato con AWS CloudTrail, un servizio che fornisce una registrazione delle azioni intraprese da un utente, un ruolo o un AWS servizio in AWS AppSync. CloudTrail acquisisce le chiamate API AWS AppSync come eventi. Le chiamate acquisite includono chiamate dalla AWS AppSync console e chiamate di codice alle operazioni AWS AppSync API. Se crei un trail, puoi abilitare la distribuzione continua di CloudTrail eventi a un bucket Amazon S3, inclusi gli eventi per. AWS AppSync Se non configuri un percorso, puoi comunque visualizzare gli eventi più recenti nella

CloudTrail console nella cronologia degli eventi. Utilizzando le informazioni raccolte da CloudTrail, puoi determinare a quale richiesta è stata inviata AWS AppSync, l'indirizzo IP da cui è stata effettuata la richiesta, chi ha effettuato la richiesta, quando è stata effettuata e dettagli aggiuntivi.

Per ulteriori informazioni CloudTrail, consulta la [Guida AWS CloudTrail per l'utente](#).

AWS AppSync informazioni in CloudTrail

CloudTrail è abilitato sul tuo AWS account al momento della creazione dell'account. Quando si verifica un'attività in AWS AppSync, tale attività viene registrata in un CloudTrail evento insieme ad altri eventi AWS di servizio nella cronologia degli eventi. Puoi visualizzare, cercare e scaricare gli eventi recenti nel tuo AWS account. Per ulteriori informazioni, consulta [Visualizzazione degli eventi con la cronologia degli CloudTrail eventi](#).

Per una registrazione continua degli eventi nel tuo AWS account, inclusi gli eventi di AWS AppSync, crea un percorso. Un trail consente di CloudTrail inviare file di log a un bucket Amazon S3. Per impostazione predefinita, quando crei un percorso nella console, il percorso si applica a tutte le AWS regioni. Il trail registra gli eventi di tutte le regioni della AWS partizione e consegna i file di log al bucket Amazon S3 specificato. Inoltre, puoi configurare altri AWS servizi per analizzare ulteriormente e agire in base ai dati sugli eventi raccolti nei log. CloudTrail Per ulteriori informazioni, consulta gli argomenti seguenti:

- [Panoramica della creazione di un percorso](#)
- [CloudTrail servizi e integrazioni supportati](#)
- [Configurazione delle notifiche Amazon SNS per CloudTrail](#)
- [Ricezione di file di CloudTrail registro da più regioni](#) e [ricezione di file di CloudTrail registro da più account](#)

AWS AppSync supporta la registrazione delle chiamate effettuate tramite l' AWS AppSync API. Al momento, le chiamate verso l'utente APIs e le chiamate effettuate ai resolver non vengono registrate. AWS AppSync CloudTrail

Ogni evento o voce di log contiene informazioni sull'utente che ha generato la richiesta. Le informazioni di identità consentono di determinare quanto segue:

- Se la richiesta è stata effettuata con credenziali utente root o AWS Identity and Access Management (IAM).

- Se la richiesta è stata effettuata con le credenziali di sicurezza temporanee per un ruolo o un utente federato.
- Se la richiesta è stata effettuata da un altro AWS servizio.

Per ulteriori informazioni, consulta [Elemento CloudTrail userIdentity](#).

Comprensione delle AWS AppSync voci dei file di registro

Un trail è una configurazione che consente la distribuzione di eventi come file di log in un bucket Amazon S3 specificato dall'utente. CloudTrail i file di registro contengono una o più voci di registro. Un evento rappresenta una singola richiesta proveniente da qualsiasi fonte e include informazioni sull'azione richiesta, la data e l'ora dell'azione, i parametri della richiesta e così via. CloudTrail i file di registro non sono una traccia ordinata dello stack delle chiamate API pubbliche, quindi non vengono visualizzati in un ordine specifico.

L'esempio seguente mostra una voce di CloudTrail registro che mostra l'GetGraphQLApiazione eseguita tramite la AWS AppSync console:

```
{
  "eventVersion": "1.08",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "ABCDEFXAMPLEPRINCIPAL:nikkiwolf",
    "arn": "arn:aws:sts::111122223333:assumed-role/admin/nikkiwolf",
    "accountId": "111122223333",
    "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
    "sessionContext": {
      "sessionIssuer": {
        "type": "Role",
        "principalId": "AIDAJ45Q7YFFAREXAMPLE",
        "arn": "arn:aws:iam::111122223333:role/admin",
        "accountId": "111122223333",
        "userName": "admin"
      },
      "webIdFederationData": {},
      "attributes": {
        "mfaAuthenticated": "false",
        "creationDate": "2021-03-12T22:41:48Z"
      }
    }
  },
}
```

```

    "eventTime": "2021-03-12T22:46:18Z",
    "eventSource": "appsync.amazonaws.com",
    "eventName": "GetGraphQLApi",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "203.0.113.69",
    "userAgent": "aws-internal/3 aws-sdk-java/1.11.942
Linux/4.9.230-0.1.ac.223.84.332.metal1.x86_64 OpenJDK_64-Bit_Server_VM/25.282-b08
java/1.8.0_282 vendor/Oracle_Corporation",
    "requestParameters": {
      "apiId": "xhxt3typtfnmidkhcexampleid"
    },
    "responseElements": null,
    "requestID": "2fc43a35-a552-4b5d-be6e-12553a03dd12",
    "eventID": "b95b0ad9-8c71-4252-a2ec-5dc2fe5f8ae8",
    "readOnly": true,
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "eventCategory": "Management",
    "recipientAccountId": "111122223333"
  }
}

```

L'esempio seguente mostra una voce di CloudTrail registro che dimostra l>CreateApiKeyazione eseguita tramite: AWS CLI

```

{
  "eventVersion": "1.08",
  "userIdentity": {
    "type": "IAMUser",
    "principalId": "ABCDEFXAMPLEPRINCIPAL",
    "arn": "arn:aws:iam::111122223333:user/nikkiwolf",
    "accountId": "111122223333",
    "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
    "userName": "nikkiwolf"
  },
  "eventTime": "2021-03-12T22:49:10Z",
  "eventSource": "appsync.amazonaws.com",
  "eventName": "CreateApiKey",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "203.0.113.69",
  "userAgent": "aws-cli/2.0.11 Python/3.7.4 Darwin/18.7.0 botocore/2.0.0dev15",
  "requestParameters": {
    "apiId": "xhxt3typtfnmidkhcexampleid"
  },
}

```

```
{
  "responseElements": {
    "apiKey": {
      "id": "****",
      "expires": 1616191200,
      "deletes": 1621375200
    }
  },
  "requestID": "e152190e-04ba-4d0a-ae7b-6bfc0bcea6af",
  "eventID": "ba3f39e0-9d87-41c5-abbb-2000abcb6013",
  "readOnly": false,
  "eventType": "AwsApiCall",
  "managementEvent": true,
  "eventCategory": "Management",
  "recipientAccountId": "111122223333"
}
```

Best practice di sicurezza per AWS AppSync

La protezione AWS AppSync è molto più che attivare alcune leve o configurare la registrazione. Nelle sezioni seguenti vengono illustrate le migliori pratiche di sicurezza che variano a seconda dell'utilizzo del servizio.

Comprendi i metodi di autenticazione

AWS AppSync offre diversi modi per autenticare gli utenti su GraphQL APIs. Ogni metodo presenta dei compromessi in termini di sicurezza, verificabilità e usabilità.

Sono disponibili i seguenti metodi di autenticazione comuni:

- I pool di utenti di Amazon Cognito consentono all'API GraphQL di utilizzare gli attributi utente per il controllo e il filtraggio granulari degli accessi.
- I token API hanno una durata limitata e sono appropriati per sistemi automatizzati, come i sistemi di integrazione continua e l'integrazione con sistemi esterni. APIs
- AWS Identity and Access Management (IAM) è appropriato per le applicazioni interne gestite nel tuo Account AWS.
- OpenID Connect consente di controllare e federare l'accesso con il protocollo OpenID Connect.

Per ulteriori informazioni sull'autenticazione e l'autorizzazione in AWS AppSync, vedere.

[Configurazione dell'autorizzazione e dell'autenticazione per proteggere GraphQL APIs](#)

Scopri come si propagano le modifiche alla configurazione delle API

Quando salvi le modifiche alla configurazione dell'API, AWS AppSync inizia a propagare le modifiche. Fino a quando la modifica alla configurazione non viene propagata, AWS AppSync continua a fornire i contenuti della configurazione precedente. Dopo la propagazione della modifica alla configurazione, inizia AWS AppSync immediatamente a pubblicare i contenuti in base alla nuova configurazione. Durante AWS AppSync la propagazione delle modifiche relative a un'API, non siamo in grado di determinare se l'API stia servendo i tuoi contenuti sulla base della configurazione precedente o della nuova configurazione.

Usa TLS per i resolver HTTP

Quando usi i resolver HTTP, assicurati di utilizzare connessioni protette da TLS (HTTPS) laddove possibile. Per un elenco completo dei certificati TLS affidabili, consulta [AWS AppSync Autorità di certificazione \(CA\) riconosciute dagli AWS AppSync endpoint HTTPS](#)

Usa ruoli con il minor numero di autorizzazioni possibile

Quando utilizzi resolver come il [resolver DynamoDB, utilizza ruoli che offrono la visualizzazione più restrittiva delle tue risorse, come le tabelle Amazon DynamoDB](#).

Le migliori pratiche in materia di policy IAM

Le politiche basate sull'identità determinano se qualcuno può creare, accedere o eliminare AWS AppSync risorse nel tuo account. Queste azioni possono comportare costi aggiuntivi per l' Account AWS. Quando si creano o modificano policy basate sull'identità, seguire queste linee guida e raccomandazioni:

- Inizia con le policy AWS gestite e passa alle autorizzazioni con privilegi minimi: per iniziare a concedere autorizzazioni a utenti e carichi di lavoro, utilizza le politiche gestite che concedono le autorizzazioni per molti casi d'uso comuni. AWS Sono disponibili nel tuo. Account AWS Ti consigliamo di ridurre ulteriormente le autorizzazioni definendo politiche gestite dai AWS clienti specifiche per i tuoi casi d'uso. Per maggiori informazioni, consulta [Policy gestite da AWS](#) o [Policy gestite da AWS per le funzioni dei processi](#) nella Guida per l'utente di IAM.
- Applicazione delle autorizzazioni con privilegio minimo - Quando si impostano le autorizzazioni con le policy IAM, concedere solo le autorizzazioni richieste per eseguire un'attività. È possibile farlo definendo le azioni che possono essere intraprese su risorse specifiche in condizioni specifiche, note anche come autorizzazioni con privilegio minimo. Per maggiori informazioni sull'utilizzo di IAM

per applicare le autorizzazioni, consulta [Policy e autorizzazioni in IAM](#) nella Guida per l'utente di IAM.

- Condizioni d'uso nelle policy IAM per limitare ulteriormente l'accesso - Per limitare l'accesso ad azioni e risorse è possibile aggiungere una condizione alle policy. Ad esempio, è possibile scrivere una condizione di policy per specificare che tutte le richieste devono essere inviate utilizzando SSL. Puoi anche utilizzare le condizioni per concedere l'accesso alle azioni del servizio se vengono utilizzate tramite uno specifico Servizio AWS, ad esempio CloudFormation. Per maggiori informazioni, consultare la sezione [Elementi delle policy JSON di IAM: condizione](#) nella Guida per l'utente di IAM.
- Utilizzo dello strumento di analisi degli accessi IAM per convalidare le policy IAM e garantire autorizzazioni sicure e funzionali - Lo strumento di analisi degli accessi IAM convalida le policy nuove ed esistenti in modo che aderiscano al linguaggio (JSON) della policy IAM e alle best practice di IAM. Lo strumento di analisi degli accessi IAM offre oltre 100 controlli delle policy e consigli utili per creare policy sicure e funzionali. Per maggiori informazioni, consultare [Convalida delle policy per il Sistema di analisi degli accessi IAM](#) nella Guida per l'utente di IAM.
- Richiedi l'autenticazione a più fattori (MFA): se hai uno scenario che richiede utenti IAM o un utente root nel Account AWS tuo, attiva l'MFA per una maggiore sicurezza. Per richiedere la MFA quando vengono chiamate le operazioni API, aggiungere le condizioni MFA alle policy. Per maggiori informazioni, consultare [Protezione dell'accesso API con MFA](#) nella Guida per l'utente di IAM.

Per maggiori informazioni sulle best practice in IAM, consulta [Best practice di sicurezza in IAM](#) nella Guida per l'utente di IAM.

AWS AppSync riferimento al resolver () JavaScript

Le seguenti sezioni contengono il riferimento al APPSYNC_JS runtime e al JavaScript resolver:

- [JavaScript panoramica sui resolver: scopri di più](#) su come funzionano i resolver. AWS AppSync
- [Riferimento agli oggetti contestuali del resolver: scopri di più sull'oggetto contestuale](#) e su come viene utilizzato nei resolver.
- [JavaScript caratteristiche di runtime per resolver e funzioni - Scopri di più sulle funzionalità di runtime supportate e](#) sull'utilizzo delle utilità per semplificare il codice.
- [JavaScript riferimento alla funzione resolver per DynamoDB - Scopri di più su come i resolver interagiscono con DynamoDB.](#)
- [JavaScript riferimento alla funzione resolver per OpenSearch](#) : scopri di più sulla struttura delle richieste e delle risposte del resolver e sulle interazioni con Service. OpenSearch
- [JavaScript riferimento alla funzione resolver per Lambda](#) - Scopri di più sulla struttura di richieste e risposte del resolver e sulle interazioni con Lambda.
- [JavaScript riferimento alla funzione resolver per l'origine EventBridge dati](#) - Scopri di più sulla struttura delle richieste e delle risposte del resolver e sulle interazioni con. EventBridge
- [JavaScript riferimento alla funzione resolver per l'origine dati None](#) - Scopri di più sulla struttura di richiesta e risposta del resolver e sulle interazioni con le fonti di dati NONE.
- [JavaScript riferimento alla funzione resolver per HTTP](#) - Scopri di più sulla struttura di richiesta e risposta del resolver e sulle interazioni con gli endpoint HTTP.
- [JavaScript riferimento alla funzione resolver per Amazon RDS](#) - Scopri di più sulla struttura del resolver e sulle interazioni con RDS.
- [JavaScript riferimento alla funzione resolver per Amazon Bedrock](#) - Scopri di più sulla struttura del resolver e sulle interazioni con Amazon Bedrock.

AWS AppSync JavaScript panoramica dei resolver

AWS AppSync consente di rispondere alle richieste GraphQL eseguendo operazioni sulle fonti di dati. Per ogni campo GraphQL su cui si desidera eseguire una query, una mutazione o una sottoscrizione, è necessario allegare un resolver.

I resolver sono i connettori tra GraphQL e una fonte di dati. Spiegano AWS AppSync come tradurre una richiesta GraphQL in entrata in istruzioni per l'origine dati di backend e come tradurre la risposta

da tale fonte di dati in una risposta GraphQL. Con AWS AppSync, puoi scrivere i tuoi resolver utilizzando JavaScript ed eseguirli nell'ambiente (). AWS AppSync APPSYNC_JS

AWS AppSync consente di scrivere resolver di unità o resolver di pipeline composti da più funzioni in una pipeline. AWS AppSync

Funzionalità di runtime supportate

Il AWS AppSync JavaScript runtime fornisce un sottoinsieme di JavaScript librerie, utilità e funzionalità. Per un elenco completo delle caratteristiche e delle funzionalità supportate dal APPSYNC_JS runtime, consultate Funzionalità di [JavaScript runtime per resolver e funzioni](#).

Risolutori di unità

Un resolver di unità è composto da codice che definisce un gestore di richieste e risposte che vengono eseguiti su un'origine dati. Il gestore delle richieste accetta un oggetto di contesto come argomento e restituisce il payload della richiesta utilizzato per chiamare l'origine dei dati. Il gestore della risposta riceve un payload dall'origine dati con il risultato della richiesta eseguita. Il gestore di risposte trasforma il payload in una risposta GraphQL per risolvere il campo GraphQL. Nell'esempio seguente, un resolver recupera un elemento da un'origine dati DynamoDB:

```
import * as ddb from '@aws-appsync/utils/dynamodb'

export function request(ctx) {
  return ddb.get({ key: { id: ctx.args.id } });
}

export const response = (ctx) => ctx.result;
```

Anatomia di un resolver di pipeline JavaScript

Un pipeline resolver è composto da codice che definisce un gestore di richieste e risposte e un elenco di funzioni. Ogni funzione dispone di un gestore di richieste e risposte che esegue su un'origine dati. Poiché un resolver di pipeline delega l'esecuzione a un elenco di funzioni, non è quindi collegato a nessuna fonte di dati. Funzioni e resolver di unità sono primitive che eseguono un'operazione su origini dati.

Gestore di richieste Pipeline Resolver

Il gestore delle richieste di un risolutore di pipeline (il passaggio precedente) consente di eseguire una logica di preparazione prima di eseguire le funzioni definite.

Elenco delle funzioni

L'elenco delle funzioni eseguite in sequenza da un resolver di pipeline. Il risultato della valutazione del gestore di richieste del resolver pipeline viene reso disponibile alla prima funzione come.

`ctx.prev.result` Ogni risultato della valutazione della funzione è disponibile per la funzione successiva come. `ctx.prev.result`

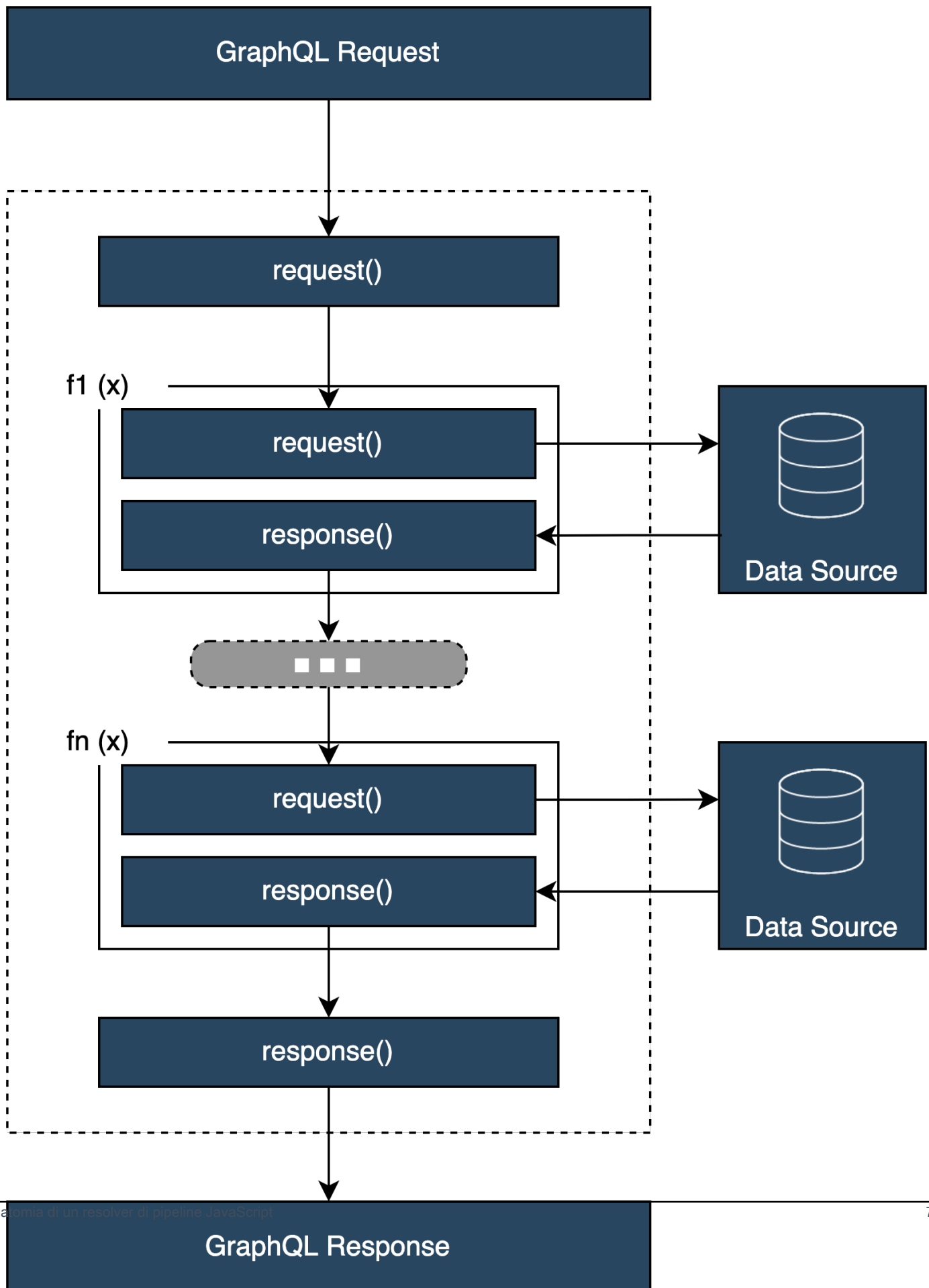
Gestore di risposte del resolver Pipeline

Il gestore di risposte di un resolver di pipeline consente di eseguire una logica finale dall'output dell'ultima funzione al tipo di campo GraphQL previsto. L'output dell'ultima funzione nell'elenco delle funzioni è disponibile nel gestore di risposte del resolver della pipeline come o. `ctx.prev.result`
`ctx.result`

Flusso di esecuzione

Dato un resolver a pipeline composto da due funzioni, l'elenco seguente rappresenta il flusso di esecuzione quando viene richiamato il resolver:

1. Gestore delle richieste del resolver Pipeline
2. Funzione 1: gestore di richieste di funzioni
3. Funzione 1: invocazione dell'origine dati
4. Funzione 1: gestore della risposta alla funzione
5. Funzione 2: gestore di richieste di funzioni
6. Funzione 2: invocazione dell'origine dati
7. Funzione 2: gestore della risposta alla funzione
8. Gestore di risposte del resolver Pipeline



Utili utilità integrate in runtime **APPSYNC_JS**

Nell'utilizzo dei resolver di pipeline, è possibile avvalersi delle seguenti funzionalità.

`ctx.stash`

Lo `stash` è un oggetto reso disponibile all'interno di ogni resolver e gestore di richieste e risposte di funzioni. La stessa istanza `stash` dura attraverso una singola esecuzione del resolver. Ciò significa che puoi utilizzare lo `stash` per passare dati arbitrari tra gestori di richieste e risposte e tra funzioni in un resolver di pipeline. Puoi testare lo `stash` come un normale oggetto. JavaScript

`ctx.prev.result`

La voce `ctx.prev.result` corrisponde al risultato dell'operazione precedentemente eseguita nella pipeline. Se l'operazione precedente era il gestore delle richieste del resolver della pipeline, allora `ctx.prev.result` viene reso disponibile alla prima funzione della catena. Se l'operazione precedente corrisponde alla prima funzione, `ctx.prev.result` è l'output della prima funzione, disponibile per la seconda funzione della pipeline. Se l'operazione precedente era l'ultima funzione, `ctx.prev.result` rappresenta l'output dell'ultima funzione e viene resa disponibile al gestore delle risposte del resolver della pipeline.

`util.error`

Con `util.error` è possibile generare un errore di campo. L'utilizzo `util.error` all'interno di un gestore di richieste o risposte di funzione genera immediatamente un errore di campo, che impedisce l'esecuzione delle funzioni successive. Per maggiori dettagli e altre `util.error` firme, visita le funzionalità di [JavaScript runtime per resolver e funzioni](#).

`util.appendError`

`util.appendError` è simile a `util.error()`, con la principale distinzione che non interrompe la valutazione del gestore. Segnala invece che c'è stato un errore nel campo, ma consente di valutare il gestore e di conseguenza di restituire i dati. L'utilizzo di `util.appendError` all'interno di una funzione non interrompe il flusso di esecuzione della pipeline. Per maggiori dettagli e altre `util.error` firme, visita le funzionalità di [JavaScript runtime per resolver e funzioni](#).

`runtime.earlyReturn`

La `runtime.earlyReturn` funzione consente di tornare prematuramente da qualsiasi funzione di richiesta. Se si utilizza `runtime.earlyReturn` all'interno di un resolver, il gestore di richieste verrà

restituito dal resolver. AWS AppSync La chiamata da un gestore di richieste di funzioni restituirà dalla funzione e continuerà l'esecuzione verso la funzione successiva nella pipeline o il gestore di risposte del resolver.

Scrittura di resolver per pipeline

Un risolutore di pipeline dispone anche di un gestore di richiesta e di risposta che circonda l'esecuzione delle funzioni nella pipeline: il suo gestore di richieste viene eseguito prima della richiesta della prima funzione e il suo gestore di risposta viene eseguito dopo la risposta dell'ultima funzione. Il gestore delle richieste del resolver può impostare i dati che devono essere utilizzati dalle funzioni nella pipeline. Il gestore della risposta del resolver è responsabile della restituzione dei dati mappati al tipo di output del campo GraphQL. Nell'esempio seguente, un gestore di richieste di resolver definisce `allowedGroups`; i dati restituiti devono appartenere a uno di questi gruppi. Questo valore può essere utilizzato dalle funzioni del resolver per richiedere dati. Il gestore delle risposte del resolver esegue un controllo finale e filtra il risultato per assicurarsi che vengano restituiti solo gli elementi che appartengono ai gruppi consentiti.

```
import { util } from '@aws-appsync/utils';

/**
 * Called before the request function of the first AppSync function in the pipeline.
 * @param ctx the context object holds contextual information about the function
 * invocation.
 */
export function request(ctx) {
  ctx.stash.allowedGroups = ['admin'];
  ctx.stash.startedAt = util.time.nowISO8601();
  return {};
}

/**
 * Called after the response function of the last AppSync function in the pipeline.
 * @param ctx the context object holds contextual information about the function
 * invocation.
 */
export function response(ctx) {
  const result = [];
  for (const item of ctx.prev.result) {
    if (ctx.stash.allowedGroups.indexOf(item.group) > -1) result.push(item);
  }
  return result;
}
```

Funzioni di scrittura AWS AppSync

AWS AppSync le funzioni consentono di scrivere una logica comune che è possibile riutilizzare su più resolver dello schema. Ad esempio, puoi avere una AWS AppSync funzione chiamata `QUERY_ITEMS` responsabile dell'interrogazione di elementi da un'origine dati Amazon DynamoDB. Per i resolver con cui desideri interrogare gli elementi, aggiungi semplicemente la funzione alla pipeline del resolver e fornisci l'indice di query da utilizzare. La logica non deve essere reimplementata.

Argomenti supplementari

Argomenti

- [Esempio di risolutore di pipeline con Amazon DynamoDB](#)
- [Configurazione delle utilità per il runtime APPSYNC_JS](#)
- [Raggruppamento e TypeScript mappe di origine per il runtime APPSYNC_JS](#)
- [Verifica del resolver e dei gestori di funzioni](#)
- [Migrazione da VTL a JavaScript](#)
- [Scelta tra accesso diretto alla fonte di dati e invio di proxy tramite un'origine dati Lambda](#)

Esempio di risolutore di pipeline con Amazon DynamoDB

Supponiamo di voler collegare un resolver di pipeline a un campo denominato `getPost(id:ID!)` che restituisce un `Post` tipo da un'origine dati Amazon DynamoDB con la seguente query GraphQL:

```
getPost(id:1){
  id
  title
  content
}
```

Innanzitutto, collega un semplice resolver a con il codice seguente. `Query.getPost` Questo è un esempio di codice resolver semplice. Non esiste una logica definita nel gestore della richiesta e il gestore della risposta restituisce semplicemente il risultato dell'ultima funzione.

```
/**
 * Invoked before the request handler of the first AppSync function in the
 * pipeline.
 * The resolver `request` handler allows to perform some preparation logic
```



```

* before executing the defined functions in your pipeline.
* @param ctx the context object holds contextual information about the function
  invocation.
*/
export function request(ctx) {
  return {}
}

/**
* Invoked **after** the response handler of the last AppSync function in the pipeline.
* The resolver `response` handler allows to perform some final evaluation logic
* from the output of the last function to the expected GraphQL field type.
* @param ctx the context object holds contextual information about the function
  invocation.
*/
export function response(ctx) {
  return ctx.prev.result
}

```

Quindi, definisci la funzione GET_ITEM che recupera un postitem dalla tua fonte di dati:

```

import { util } from '@aws-appsync/utils'
import * as ddb from '@aws-appsync/utils/dynamodb'

/**
* Request a single item from the attached DynamoDB table datasource
* @param ctx the context object holds contextual information about the function
  invocation.
*/
export function request(ctx) {
  const { id } = ctx.args
  return ddb.get({ key: { id } })
}

/**
* Returns the result
* @param ctx the context object holds contextual information about the function
  invocation.
*/
export function response(ctx) {
  const { error, result } = ctx
  if (error) {
    return util.appendError(error.message, error.type, result)
  }
}

```

```

    }
    return ctx.result
  }

```

Se si verifica un errore durante la richiesta, il gestore di risposte della funzione aggiunge un errore che verrà restituito al client chiamante nella risposta GraphQL. Aggiungi la `GET_ITEM` funzione all'elenco delle funzioni del resolver. Quando si esegue la query, il gestore delle richieste della `GET_ITEM` funzione utilizza gli strumenti forniti dal modulo AWS AppSync DynamoDB per creare una `DynamoDBGetItem` richiesta utilizzando come chiave. `id` `ddb.get({ key: { id } })` genera l'operazione appropriata: `GetItem`

```

{
  "operation" : "GetItem",
  "key" : {
    "id" : { "S" : "1" }
  }
}

```

AWS AppSync utilizza la richiesta per recuperare i dati da Amazon DynamoDB. Una volta restituiti, i dati vengono gestiti dal gestore delle risposte della `GET_ITEM` funzione, che verifica la presenza di errori e quindi restituisce il risultato.

```

{
  "result" : {
    "id": 1,
    "title": "hello world",
    "content": "<long story>"
  }
}

```

Infine, il gestore di risposte del resolver restituisce direttamente il risultato.

Lavorare con gli errori

Se si verifica un errore nella funzione durante una richiesta, l'errore verrà reso disponibile nel gestore della risposta alla funzione `inctx.error`. È possibile aggiungere l'errore alla risposta GraphQL utilizzando `util.appendError` utilità. È possibile rendere l'errore disponibile per altre funzioni della pipeline utilizzando lo stash. Vedere l'esempio di seguito.

```

/**

```

```
* Returns the result
* @param ctx the context object holds contextual information about the function
  invocation.
*/
export function response(ctx) {
  const { error, result } = ctx;
  if (error) {
    if (!ctx.stash.errors) ctx.stash.errors = []
    ctx.stash.errors.push(ctx.error)
    return util.appendError(error.message, error.type, result);
  }
  return ctx.result;
}
```

Configurazione delle utilità per il runtime **APPSYNC_JS**

AWS AppSync fornisce due librerie che facilitano lo sviluppo di resolver con il runtime: APPSYNC_JS

- `@aws-appsync/eslint-plugin`- Rileva e corregge rapidamente i problemi durante lo sviluppo.
- `@aws-appsync/utils`- Fornisce la convalida dei tipi e il completamento automatico negli editor di codice.

Configurazione del plugin eslint

[ESLint](#) è uno strumento che analizza staticamente il codice per trovare rapidamente i problemi. Puoi eseguirlo ESLint come parte della tua pipeline di integrazione continua. `@aws-appsync/eslint-plugin` è un ESLint plugin che rileva la sintassi non valida nel codice quando sfrutta il runtime. APPSYNC_JS Il plugin consente di ottenere rapidamente un feedback sul codice durante lo sviluppo senza dover inviare le modifiche al cloud.

`@aws-appsync/eslint-plugin` fornisce due set di regole che è possibile utilizzare durante lo sviluppo.

«plugin: `@aws -appsync/base`» configura un set di regole di base che puoi sfruttare nel tuo progetto:

Regola	Description
no-async	I processi e le promesse asincroni non sono supportati.

Regola	Description
senza attesa	I processi e le promesse asincroni non sono supportati.
nessuna classe	Le classi non sono supportate.
no per	for non è supportato (ad eccezione di for-in e for-of, che sono supportati)
no-continue	continue non è supportato.
senza generatori	I generatori non sono supportati.
senza rendimento	yield non è supportato.
senza etichette	Le etichette non sono supportate.
no-questo	this la parola chiave non è supportata.
nessun tentativo	La struttura TRY/catch non è supportata.
nel giro di poco	I loop While non sono supportati.
no-disallowed-unary-operators	++ --, e gli operatori ~ unari non sono consentiti.
no-disallowed-binary-operators	L'instanceof operatore non è consentito.
nessuna promessa	I processi e le promesse asincroni non sono supportati.

«plugin: @aws -appsync/recommended» fornisce alcune regole aggiuntive ma richiede anche l'aggiunta di configurazioni al progetto. TypeScript

Regola	Description
nessuna ricorsione	Le chiamate di funzioni ricorsive non sono consentite.

Regola	Description
no-disallowed-methods	Alcuni metodi non sono consentiti. Vedi il riferimento per un set completo di funzioni integrate supportate.
no-function-passing	Non è consentito passare funzioni come argomenti di funzione alle funzioni.
no-function-reassign	Le funzioni non possono essere riassegnate.
no-function-return	Le funzioni non possono essere il valore restituito dalle funzioni.

Per aggiungere il plugin al tuo progetto, segui i passaggi di installazione e utilizzo in [Getting Started with ESLint](#). Quindi, installa il [plugin](#) nel tuo progetto usando il gestore dei pacchetti del progetto (ad esempio, npm, yarn o pnpm):

```
$ npm install @aws-appsync/eslint-plugin
```

Nel tuo `.eslintrc.{js,yml,json}` file, aggiungi «plugin: @aws -appsync/base» o «plugin: @aws -appsync/recommended» alla proprietà. `extends` Lo snippet riportato di seguito è un esempio di configurazione di base per: `.eslintrc JavaScript`

```
{
  "extends": ["plugin:@aws-appsync/base"]
}
```

Per utilizzare il set di regole «plugin: @aws -appsync/recommended», installa la dipendenza richiesta:

```
$ npm install -D @typescript-eslint/parser
```

Quindi, crea un file: `.eslintrc.js`

```
{
  "parser": "@typescript-eslint/parser",
  "parserOptions": {
    "ecmaVersion": 2018,
```

```
"project": "./tsconfig.json"
},
"extends": ["plugin:@aws-appsync/recommended"]
}
```

Raggruppamento TypeScript e mappe di origine per il runtime **APPSYNC_JS**

TypeScript migliora AWS AppSync lo sviluppo garantendo la sicurezza dei tipi e il rilevamento precoce degli errori. È possibile scrivere TypeScript codice localmente e traspirarlo JavaScript prima di utilizzarlo con il runtime. APPSYNC_JS Il processo inizia con l'installazione TypeScript e la configurazione di tsconfig.json per l'ambiente. APPSYNC_JS È quindi possibile utilizzare strumenti di raggruppamento come esbuild per compilare e raggruppare il codice. L'Amplify CLI genererà tipi dallo schema GraphQL, consentendoti di utilizzare questi tipi nel codice resolver.

È possibile utilizzare librerie personalizzate ed esterne nel codice del resolver e della funzione, purché soddisfino i requisiti. APPSYNC_JS Gli strumenti di raggruppamento combinano il codice in un unico file da utilizzare. AWS AppSyncÈ possibile includere mappe di origine per facilitare il debug.

Sfruttare le librerie e raggruppare il codice

Nel codice del resolver e della funzione, puoi sfruttare sia le librerie personalizzate che quelle esterne purché soddisfino i requisiti. APPSYNC_JS In questo modo è possibile riutilizzare il codice esistente nell'applicazione. Per utilizzare librerie definite da più file, è necessario utilizzare uno strumento di raggruppamento, come [esbuild](#), per combinare il codice in un unico file che può quindi essere salvato nel AWS AppSync resolver o nella funzione.

Quando raggruppate il codice, tenete presente quanto segue:

- APPSYNC_JSsupporta solo ECMAScript moduli (ESM).
- @aws-appsync/*i moduli sono integrati nel APPSYNC_JS codice e non devono essere inclusi nel codice.
- L'ambiente APPSYNC_JS di runtime è simile a NodeJS in quanto il codice non viene eseguito in un ambiente browser.
- È possibile includere una mappa di origine opzionale. Tuttavia, non includere il contenuto di origine.

Per ulteriori informazioni sulle mappe di origine, consulta [Utilizzo delle mappe di origine](#).

Ad esempio, per raggruppare il codice del resolver che si trova in `src/appsync/getPost.resolver.js`, è possibile utilizzare il seguente comando ESbuild CLI:

```
$ esbuild --bundle \  
--sourcemap=inline \  
--sources-content=false \  
--target=esnext \  
--platform=node \  
--format=esm \  
--external:@aws-appsync/utils \  
--outdir=out/appsync \  
src/appsync/getPost.resolver.js
```

Creazione del codice e utilizzo di TypeScript

[TypeScript](#) è un linguaggio di programmazione sviluppato da Microsoft che offre tutte JavaScript le funzionalità insieme al sistema di TypeScript digitazione. Puoi usarlo TypeScript per scrivere codice type-safe e catturare errori e bug in fase di compilazione prima di salvare il codice in. AWS AppSync Il @aws-appsync/utils pacchetto è completamente digitato.

Il APPSYNC_JS runtime non supporta TypeScript direttamente. È necessario prima trascrivere il TypeScript codice in JavaScript codice supportato dal APPSYNC_JS runtime prima di salvarlo in. AWS AppSync Puoi usarlo TypeScript per scrivere il codice nell'ambiente di sviluppo integrato (IDE) locale, ma tieni presente che non puoi creare TypeScript codice nella AWS AppSync console.

Per iniziare, assicurati di averlo [TypeScript](#) installato nel tuo progetto. Quindi, configura le impostazioni di TypeScript transcompilazione in modo che funzionino con il APPSYNC_JS runtime utilizzando. [TSConfig](#) Ecco un esempio di tsconfig.json file di base che puoi usare:

```
// tsconfig.json  
{  
  "compilerOptions": {  
    "target": "esnext",  
    "module": "esnext",  
    "noEmit": true,  
    "moduleResolution": "node",  
  }  
}
```

È quindi possibile utilizzare uno strumento di raggruppamento come esbuild per compilare e raggruppare il codice. Ad esempio, dato un progetto il cui AWS AppSync codice si trova in src/appsync, è possibile utilizzare il seguente comando per compilare e raggruppare il codice:

```
$ esbuild --bundle \  
--sourcemap=inline \  
--sources-content=false \  
--target=esnext \  
--platform=node \  
--format=esm \  
--external:@aws-appsync/utils \  
--outdir=out/appsync \  
src/appsync/getPost.resolver.js
```

```
--sourcemap=inline \  
--sources-content=false \  
--target=esnext \  
--platform=node \  
--format=esm \  
--external:@aws-appsync/utils \  
--outdir=out/appsync \  
src/appsync/**/*.ts
```

Usare il codegen Amplify

Puoi usare la CLI [Amplify](#) per generare i tipi per il tuo schema. Dalla directory in cui si trova il `schema.graphql` file, esegui il comando seguente e consulta le istruzioni per configurare il codegen:

```
$ npx @aws-amplify/cli codegen add
```

Per rigenerare il codegen in determinate circostanze (ad esempio, quando lo schema viene aggiornato), esegui il comando seguente:

```
$ npx @aws-amplify/cli codegen
```

È quindi possibile utilizzare i tipi generati nel codice del resolver. Ad esempio, dato lo schema seguente:

```
type Todo {  
  id: ID!  
  title: String!  
  description: String  
}  
  
type Mutation {  
  createTodo(title: String!, description: String): Todo  
}  
  
type Query {  
  listTodos: Todo  
}
```

È possibile utilizzare i tipi generati nella seguente AWS AppSync funzione di esempio:


```
import { Context, util } from '@aws-appsync/utils'
import * as ddb from '@aws-appsync/utils/dynamodb'
import { CreateTodoMutationVariables, Todo } from './API' // codegen

export function request(ctx: Context<CreateTodoMutationVariables>) {
  ctx.args.description = ctx.args.description ?? 'created on ' + util.time.nowISO8601()
  return ddb.put<Todo>({ key: { id: util.autoId() }, item: ctx.args })
}

export function response(ctx) {
  return ctx.result as Todo
}
```

Utilizzo di farmaci generici in TypeScript

È possibile utilizzare i farmaci generici con diversi dei tipi forniti. Ad esempio, lo snippet seguente è di un tipo: `Todo`

```
export type Todo = {
  __typename: "Todo",
  id: string,
  title: string,
  description?: string | null,
};
```

È possibile scrivere un resolver per un abbonamento che utilizza. `Todo` Nel tuo IDE, le definizioni dei tipi e i suggerimenti di completamento automatico ti guideranno nell'utilizzo corretto dell'utilità di `toSubscriptionFilter` trasformazione:

```
import { util, Context, extensions } from '@aws-appsync/utils'
import { Todo } from './API'

export function request(ctx: Context) {
  return {}
}

export function response(ctx: Context) {
  const filter = util.transform.toSubscriptionFilter<Todo>({
    title: { beginsWith: 'hello' },
    description: { contains: 'created' },
  })
}
```

```
extensions.setSubscriptionFilter(filter)
return null
}
```

Elencare i pacchetti

Puoi aggiungere automaticamente il linking ai tuoi pacchetti importando il plugin. `esbuild-plugin-eslint` È quindi possibile abilitarlo fornendo un `plugins` valore che abiliti le funzionalità di `eslint`. Di seguito è riportato uno snippet che utilizza l' `JavaScript API esbuild` in un file chiamato: `build.mjs`

```
/* eslint-disable */
import { build } from 'esbuild'
import eslint from 'esbuild-plugin-eslint'
import glob from 'glob'
const files = await glob('src/**/*.ts')

await build({
  format: 'esm',
  target: 'esnext',
  platform: 'node',
  external: ['@aws-appsync/utils'],
  outdir: 'dist/',
  entryPoints: files,
  bundle: true,
  plugins: [eslint({ useEslintrc: true })],
})
```

Utilizzo delle mappe di origine

Puoi fornire una mappa sorgente in linea (`sourcemap`) con il tuo `JavaScript` codice. Le mappe di origine sono utili quando si crea un pacchetto `JavaScript` o si `TypeScript` codifica e si desidera visualizzare i riferimenti ai file sorgente di input nei log e nei messaggi di errore di runtime `JavaScript`.

Il tuo `sourcemap` deve apparire alla fine del codice. È definito da una singola riga di commento che segue il seguente formato:

```
//# sourceMappingURL=data:application/json;base64,<base64 encoded string>
```

Ecco un esempio:

```
//# sourceMappingURL=data:application/
json;base64,ewogICJ2ZXJzaW9uIjojMywKICAic291cmNlcyI6IFsibGliLmpzIiwgImNvZGUuanMiXSswKICAibW9wFwcGGL
```

Le mappe di origine possono essere create con esbuild. L'esempio seguente mostra come utilizzare l'JavaScriptAPI esbuild per includere una mappa sorgente in linea quando il codice viene creato e raggruppato:

```
/* eslint-disable */
import { build } from 'esbuild'
import eslint from 'esbuild-plugin-eslint'
import glob from 'glob'
const files = await glob('src/**/*.ts')

await build({
  sourcemap: 'inline',
  sourcesContent: false,

  format: 'esm',
  target: 'esnext',
  platform: 'node',
  external: ['@aws-appsync/utils'],
  outdir: 'dist/',
  entryPoints: files,
  bundle: true,
  plugins: [eslint({ useEslintrc: true })],
})
```

In particolare, le `sourcesContent` opzioni `sourcemap` and specificano che una mappa di origine deve essere aggiunta in linea alla fine di ogni build ma non deve includere il contenuto sorgente. Per convenzione, consigliamo di non includere il contenuto sorgente nel tuo `sourcemap`. Puoi disabilitarlo in esbuild impostando `sources-content` su `false`.

Per illustrare come funzionano le mappe di origine, guarda il seguente esempio in cui un codice resolver fa riferimento alle funzioni di supporto di una libreria di supporto. Il codice contiene istruzioni di registro nel codice del resolver e nella libreria helper:

`./src/default.resolver.ts` (il tuo resolver)

```
import { Context } from '@aws-appsync/utils'
import { hello, logit } from './helper'
```

```
export function request(ctx: Context) {
  console.log('start >')
  logit('hello world', 42, true)
  console.log('< end')
  return 'test'
}

export function response(ctx: Context): boolean {
  hello()
  return ctx.prev.result
}
```

`.src/helper.ts` (un file di supporto)

```
export const logit = (...rest: any[]) => {
  // a special logger
  console.log('[logger]', ...rest.map((r) => `<${r}>`))
}

export const hello = () => {
  // This just returns a simple sentence, but it could do more.
  console.log('i just say hello..')
}
```

Quando create e raggruppate il file resolver, il codice del resolver includerà una mappa sorgente in linea. Quando il resolver è in esecuzione, nei log vengono visualizzate le seguenti voci: CloudWatch

```
INFO - ../src/default.resolver.ts:5:2: "start >"
INFO - ../src/helper.ts:3:2: "[logger]" "<hello world>" "<42>" "<true>"
INFO - ../src/default.resolver.ts:7:2: "< end"

{"logType": "BeforeRequestFunctionEvaluation", "path": ["logstuff"], "fieldName": "logstuff", "resolverArn": "arn:aws:
INFO - ../src/helper.ts:8:2: "i just say hello.."

{"logType": "AfterResponseFunctionEvaluation", "path": ["logstuff"], "fieldName": "logstuff", "resolverArn": "arn:aws:
```

Esaminando le voci del CloudWatch registro, noterai che le funzionalità dei due file sono state raggruppate e vengono eseguite contemporaneamente. Il nome originale di ogni file si riflette chiaramente anche nei log.

Verifica del resolver e dei gestori di funzioni in AWS AppSync

È possibile utilizzare il comando `EvaluateCode` API per testare in remoto il resolver e i gestori di funzioni con dati simulati prima di salvare il codice su un resolver o su una funzione.

Per iniziare a usare il comando, assicurati di aver aggiunto l'autorizzazione alla tua policy.
 appsync:evaluatecode Esempio:

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "appsync:evaluateCode",
      "Resource": "arn:aws:appsync:us-east-1:111122223333:*"
    }
  ]
}
```

È possibile sfruttare il comando utilizzando la [AWS CLI](#) o [AWS SDKs](#). Ad esempio, per testare il codice utilizzando la CLI, è sufficiente puntare al file, fornire un contesto e specificare il gestore che si desidera valutare:

```
aws appsync evaluate-code \
  --code file://code.js \
  --function request \
  --context file://context.json \
  --runtime name=APPSYNC_JS,runtimeVersion=1.0.0
```

La risposta contiene un file `evaluationResult` contenente il payload restituito dal gestore. Contiene anche un `logs` oggetto che contiene l'elenco dei log generati dal gestore durante la valutazione. Ciò semplifica il debug dell'esecuzione del codice e la visualizzazione delle informazioni sulla valutazione per facilitare la risoluzione dei problemi. Esempio:

```
{
  "evaluationResult": "{\"operation\":\"PutItem\", \"key\":{\"id\":{\"S\":\"record-id\"}}, \"attributeValues\":{\"owner\":{\"S\":\"John doe\"}, \"expectedVersion\":{\"N\":\"2\"}, \"authorId\":{\"S\":\"Sammy Davis\"}}}",
  "logs": [
    "INFO - code.js:5:3: \"current id\" \"record-id\"",
    "INFO - code.js:9:3: \"request evaluated\""
  ]
}
```

```
}
```

Il risultato della valutazione può essere analizzato come JSON, il che fornisce:

```
{
  "operation": "PutItem",
  "key": {
    "id": {
      "S": "record-id"
    }
  },
  "attributeValues": {
    "owner": {
      "S": "John doe"
    },
    "expectedVersion": {
      "N": 2
    },
    "authorId": {
      "S": "Sammy Davis"
    }
  }
}
```

Utilizzando l'SDK, puoi incorporare facilmente i test della tua suite di test per convalidare il comportamento del codice. Il nostro esempio qui utilizza il [Jest Testing Framework](#), ma qualsiasi suite di test funziona. Il seguente frammento mostra un'ipotetica esecuzione di convalida. Nota che ci aspettiamo che la risposta di valutazione sia un codice JSON valido, quindi lo utilizziamo `JSON.parse` per recuperare JSON dalla stringa di risposta:

```
const AWS = require('aws-sdk')
const fs = require('fs')
const client = new AWS.AppSync({ region: 'us-east-2' })
const runtime = {name:'APPSYNC_JS',runtimeVersion:'1.0.0'}

test('request correctly calls DynamoDB', async () => {
  const code = fs.readFileSync('./code.js', 'utf8')
  const context = fs.readFileSync('./context.json', 'utf8')
  const contextJSON = JSON.parse(context)

  const response = await client.evaluateCode({ code, context, runtime, function:
    'request' }).promise()
```

```
const result = JSON.parse(response.evaluationResult)

expect(result.key.id.S).toBeDefined()
expect(result.attributeValues.firstname.S).toEqual(contextJSON.arguments.firstname)
})
```

Ciò produce il seguente risultato:

```
Ran all test suites.
> jest

PASS ./index.test.js
# request correctly calls DynamoDB (543 ms)
Test Suites: 1 passed, 1 total
Tests: 1 passed, 1 total
Snapshots: 0 totalTime: 1.511 s, estimated 2 s
```

Migrazione da VTL a in JavaScript AWS AppSync

AWS AppSync consente di scrivere la logica aziendale per i resolver e le funzioni utilizzando VTL o JavaScript. Con entrambi i linguaggi, si scrive una logica che istruisce il AWS AppSync servizio su come interagire con le fonti di dati. Con VTL, si scrivono modelli di mappatura che devono restituire una stringa con codifica JSON valida. Con JavaScript, si scrivono gestori di richieste e risposte che restituiscono oggetti. Non restituisci una stringa con codifica JSON.

Ad esempio, utilizza il seguente modello di mappatura VTL per ottenere un elemento Amazon DynamoDB:

```
{
  "operation": "GetItem",
  "key": {
    "id": $util.dynamodb.toDynamoDBJson($ctx.args.id),
  }
}
```

L'utilità `$util.dynamodb.toDynamoDBJson` restituisce una stringa con codifica JSON. Se `$ctx.args.id` è impostato su `<id>`, il modello restituisce una stringa con codifica JSON valida:

```
{
  "operation": "GetItem",
  "key": {
```

```

      "id": {"S": "<id>"},
    }
  }
}

```

Quando si lavora con JavaScript, non è necessario stampare stringhe non elaborate con codifica JSON all'interno del codice e non è necessario utilizzare un'utilità simile. `toDynamoDBJson` Un esempio equivalente del modello di mappatura precedente è:

```

import { util } from '@aws-appsync/utils';
export function request(ctx) {
  return {
    operation: 'GetItem',
    key: {id: util.dynamodb.toDynamoDB(ctx.args.id)}
  };
}

```

Un'alternativa consiste nell'utilizzare `util.dynamodb.toMapValues`, che è l'approccio consigliato per gestire un oggetto di valori:

```

import { util } from '@aws-appsync/utils';
export function request(ctx) {
  return {
    operation: 'GetItem',
    key: util.dynamodb.toMapValues({ id: ctx.args.id }),
  };
}

```

Ciò equivale a:

```

{
  "operation": "GetItem",
  "key": {
    "id": {
      "S": "<id>"
    }
  }
}

```

Note

Consigliamo di utilizzare il modulo `DynamoDB` con sorgenti dati `DynamoDB`:


```
import * as ddb from '@aws-appsync/utils/dynamodb'

export function request(ctx) {
  ddb.get({ key: { id: ctx.args.id } })
}
```

Come altro esempio, prendi il seguente modello di mappatura per inserire un elemento in un'origine dati Amazon DynamoDB:

```
{
  "operation" : "PutItem",
  "key" : {
    "id": $util.dynamodb.toDynamoDBJson($util.autoId()),
  },
  "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}
```

Una volta valutata, questa stringa del modello di mappatura deve produrre una stringa con codifica JSON valida. Quando si utilizza JavaScript, il codice restituisce direttamente l'oggetto della richiesta:

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { id = util.autoId(), ...values } = ctx.args;
  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({ id }),
    attributeValues: util.dynamodb.toMapValues(values),
  };
}
```

che restituisce:

```
{
  "operation": "PutItem",
  "key": {
    "id": { "S": "2bff3f05-ff8c-4ed8-92b4-767e29fc4e63" }
  },
  "attributeValues": {
```

```
"firstname": { "S": "Shaggy" },
"age": { "N": 4 }
}
}
```

Note

Consigliamo di utilizzare il modulo DynamoDB con sorgenti dati DynamoDB:

```
import { util } from '@aws-appsync/utils'
import * as ddb from '@aws-appsync/utils/dynamodb'

export function request(ctx) {
  const { id = util.autoId(), ...item } = ctx.args
  return ddb.put({ key: { id }, item })
}
```

Scelta tra accesso diretto alla fonte di dati e invio di proxy tramite un'origine dati Lambda

Con AWS AppSync and the APPSYNC_JS runtime, puoi scrivere il tuo codice che implementa la tua logica aziendale personalizzata utilizzando AWS AppSync funzioni per accedere alle tue fonti di dati. Ciò semplifica l'interazione diretta con fonti di dati come Amazon DynamoDB, Aurora OpenSearch Serverless, Service APIs, HTTP AWS e altri servizi senza dover implementare servizi o infrastrutture di calcolo aggiuntivi. AWS AppSync semplifica inoltre l'interazione con una AWS Lambda funzione configurando un'origine dati Lambda. Le sorgenti dati Lambda consentono di eseguire logiche aziendali complesse utilizzando le funzionalità complete AWS Lambda di cui dispone per risolvere una richiesta GraphQL. Nella maggior parte dei casi, una AWS AppSync funzione connessa direttamente alla fonte di dati di destinazione fornirà tutte le funzionalità necessarie. In situazioni in cui è necessario implementare una logica aziendale complessa non supportata dal APPSYNC_JS runtime, è possibile utilizzare un'origine dati Lambda come proxy per interagire con l'origine dati di destinazione.

Integrazione diretta delle fonti
di dati

Fonte dati Lambda come
proxy

Caso d'uso	AWS AppSync le funzioni interagiscono direttamente con le fonti di dati API.	AWS AppSync le funzioni chiamano Lambda che interagiscono con le fonti di dati API.
Runtime	<i>APPSYNC_JS</i> (JavaScript)	Qualsiasi runtime Lambda supportato
Dimensione massima del codice	32.000 caratteri per funzione AWS AppSync	50 MB (zippato, per il caricamento diretto) per Lambda
Moduli esterni	Limitato: solo funzionalità supportate da <i>APPSYNC_JS</i>	Sì
AWS Chiama qualsiasi servizio	Sì, utilizzando l'origine dati AWS AppSync HTTP	Sì, utilizzando SDK AWS
Accesso all'intestazione della richiesta	Sì	Sì
Accesso alla rete	No	Sì
Accesso al file system	No	Sì
Registrazione e metriche	Sì	Sì
Crea e testa interamente all'interno AppSync	Sì	No
Avvio a freddo	No	No, con concorrenza preimpostata
Dimensionamento automatico	Sì, in modo trasparente tramite AWS AppSync	Sì, come configurato in Lambda
Prezzi	Nessun costo aggiuntivo	Addebitato per l'utilizzo di Lambda

AWS AppSync le funzioni che si integrano direttamente con la fonte di dati di destinazione sono ideali per casi d'uso come i seguenti:

- Interazione con Amazon DynamoDB, Aurora Serverless e Service OpenSearch
- Interazione con HTTP e passaggio di intestazioni in entrata APIs
- Interazione con AWS i servizi utilizzando sorgenti dati HTTP (con firma AWS AppSync automatica delle richieste con il ruolo di origine dati fornito)
- Implementazione del controllo dell'accesso prima di accedere alle fonti di dati
- Implementazione del filtraggio dei dati recuperati prima di soddisfare una richiesta
- Implementazione di un'orchestrazione semplice con esecuzione sequenziale di funzioni in una pipeline di resolver AWS AppSync
- Controllo delle connessioni di caching e sottoscrizione nelle query e nelle mutazioni.

AWS AppSync le funzioni che utilizzano un'origine dati Lambda come proxy sono ideali per casi d'uso come i seguenti:

- Utilizzo di un linguaggio diverso JavaScript da Velocity Template Language (VTL)
- Regolazione e controllo della CPU o della memoria per ottimizzare le prestazioni
- Importazione di librerie di terze parti o richiesta di funzionalità non supportate in APPSYNC_JS
- Effettuare più richieste di rete, and/or ottenere l'accesso al file system per soddisfare una richiesta
- Richieste in batch utilizzando la configurazione in [batch](#).

AWS AppSync JavaScript riferimento all'oggetto contestuale del resolver

AWS AppSync definisce un insieme di variabili e funzioni per lavorare con i gestori di richieste e risposte. Ciò semplifica le operazioni logiche sui dati con GraphQL. Questo documento descrive tali funzioni e fornisce esempi.

Accesso a **context**

L'oggetto `context` di un gestore di richieste e risposte è un oggetto che contiene tutte le informazioni contestuali per la chiamata del resolver. Ha la struttura seguente:

```
type Context = {
```

```
arguments: any;  
args: any;  
identity: Identity;  
source: any;  
error?: {  
  message: string;  
  type: string;  
};  
stash: any;  
result: any;  
prev: any;  
request: Request;  
info: Info;  
};
```

Note

Scoprirete spesso che l'contextoggetto viene chiamato. ctx

Ogni campo dell'contextoggetto è definito come segue:

context campi

arguments

Una mappa contenente tutti gli argomenti GraphQL per questo campo.

identity

Un oggetto contenente le informazioni sul chiamante. Vedi [Identità](#) per ulteriori informazioni sulla struttura di questo campo.

source

Una mappa contenente la risoluzione del campo padre.

stash

Lo stash è un oggetto reso disponibile all'interno di ogni resolver e gestore di funzioni. Lo stesso oggetto stash vive durante una singola esecuzione del resolver. Ciò significa che è possibile utilizzare lo stash per passare dati arbitrari tra gestori di richieste e risposte e tra funzioni in un resolver di pipeline.

Note

Non è possibile eliminare o sostituire l'intera scorta, ma è possibile aggiungere, aggiornare, eliminare e leggere le proprietà della scorta.

Puoi aggiungere elementi alla scorta modificando uno degli esempi di codice seguenti:

```
//Example 1
ctx.stash.newItem = { key: "something" }

//Example 2
Object.assign(ctx.stash, {key1: value1, key2: value})
```

Puoi rimuovere elementi dalla scorta modificando il codice seguente:

```
delete ctx.stash.key
```

result

Un container per i risultati di questo resolver. Questo campo è disponibile solo per i gestori di risposte.

Ad esempio, se stai risolvendo il `author` campo della seguente query:

```
query {
  getPost(id: 1234) {
    postId
    title
    content
    author {
      id
      name
    }
  }
}
```

Quindi la `context` variabile completa è disponibile quando viene valutato un gestore di risposte:

```
{
```

```
"arguments" : {
  id: "1234"
},
"source": {},
"result" : {
  "postId": "1234",
  "title": "Some title",
  "content": "Some content",
  "author": {
    "id": "5678",
    "name": "Author Name"
  }
},
"identity" : {
  "sourceIp" : ["x.x.x.x"],
  "userArn" : "arn:aws:iam::123456789012:user/appsync",
  "accountId" : "666666666666",
  "user" : "AIDAAAAAAAAAAAAAAAAAAAA"
}
}
```

prev.result

Il risultato di qualsiasi operazione precedente sia stata eseguita in un resolver di pipeline.

Se l'operazione precedente era il gestore delle richieste del risolutore di pipeline, allora `ctx.prev.result` rappresenta il risultato della valutazione e viene reso disponibile alla prima funzione della pipeline.

Se l'operazione precedente era la prima funzione, `ctx.prev.result` rappresenta il risultato della valutazione del gestore della risposta della prima funzione e viene reso disponibile alla seconda funzione nella pipeline.

Se l'operazione precedente era l'ultima funzione, `ctx.prev.result` rappresenta il risultato della valutazione dell'ultima funzione e viene resa disponibile al gestore di risposte del risolutore di pipeline.

info

Un oggetto contenente le informazioni sulla richiesta GraphQL. Per la struttura di questo campo, consulta [Info](#).

Identità

La sezione `identity` contiene le informazioni sul chiamante. La forma di questa sezione dipende dal tipo di autorizzazione dell'API. AWS AppSync

Per ulteriori informazioni sulle opzioni di AWS AppSync sicurezza, consulta [Autorizzazione e autenticazione](#).

Autorizzazione **API_KEY**

Il `identity` campo non è compilato.

Autorizzazione **AWS_LAMBDA**

`identity` ha la forma seguente:

```
type AppSyncIdentityLambda = {  
  resolverContext: any;  
};
```

`identity` contiene la `resolverContext` chiave, contenente lo stesso `resolverContext` contenuto restituito dalla funzione Lambda che autorizza la richiesta.

Autorizzazione **AWS_IAM**

`identity` ha la seguente forma:

```
type AppSyncIdentityIAM = {  
  accountId: string;  
  cognitoIdentityPoolId: string;  
  cognitoIdentityId: string;  
  sourceIp: string[];  
  username: string;  
  userArn: string;  
  cognitoIdentityAuthType: string;  
  cognitoIdentityAuthProvider: string;  
};
```

Autorizzazione **AMAZON_COGNITO_USER_POOLS**

`identity` ha la forma seguente:

```
type AppSyncIdentityCognito = {
```



```
sourceIp: string[];  
username: string;  
groups: string[] | null;  
sub: string;  
issuer: string;  
claims: any;  
defaultAuthStrategy: string;  
};
```

Ogni campo è definito nel modo seguente:

accountId

L'ID dell' AWS account del chiamante.

claims

Le attestazioni dell'utente.

cognitoIdentityAuthType

Autenticato o non autenticato in base al tipo di identità.

cognitoIdentityAuthProvider

Un elenco separato da virgole di informazioni sul provider di identità esterno utilizzato per ottenere le credenziali utilizzate per firmare la richiesta.

cognitoIdentityId

L'ID identificativo Amazon Cognito del chiamante.

cognitoIdentityPoolId

L'ID del pool di identità di Amazon Cognito associato al chiamante.

defaultAuthStrategy

La strategia di autorizzazione predefinita per questo chiamante (ALLOW o DENY).

issuer

L'emittente del token.

sourceIp

L'indirizzo IP di origine del chiamante che riceve. AWS AppSync Se la richiesta non include l'`x-forwarded-for` intestazione, il valore IP di origine contiene solo un singolo indirizzo IP dalla

connessione TCP. Se la richiesta include un'intestazione `x-forwarded-for`, l'IP di origine sarà un elenco di indirizzi IP dell'intestazione `x-forwarded-for` oltre all'indirizzo IP proveniente dalla connessione TCP.

sub

l'UUID dell'utente autenticato.

user

L'utente IAM.

userArn

L'Amazon Resource Name (ARN) dell'utente IAM.

username

Il nome dell'utente autenticato. In caso di autorizzazione `AMAZON_COGNITO_USER_POOLS`, il valore del nome utente è il valore dell'attributo `cognito:username`. In caso di `AWS_IAM` autorizzazione, il valore del nome utente è il valore del principale AWS utente. Se utilizzi l'autorizzazione IAM con credenziali fornite dai pool di identità di Amazon Cognito, ti consigliamo di utilizzare `cognitoIdentityId`.

Intestazioni delle richieste di accesso

AWS AppSync supporta il passaggio di intestazioni personalizzate dai client e l'accesso ad esse nei resolver GraphQL utilizzando `ctx.request.headers`. È quindi possibile utilizzare i valori dell'intestazione per azioni come l'inserimento di dati in una fonte di dati o i controlli di autorizzazione. È possibile utilizzare intestazioni di richiesta singole o multiple utilizzando `$curl` una chiave API dalla riga di comando, come illustrato negli esempi seguenti:

Esempio di intestazione singola

Supponi di impostare un'intestazione custom con il valore `nadia` come segue:

```
curl -XPOST -H "Content-Type:application/graphql" -H "custom:nadia" -H "x-api-key:<API-KEY-VALUE>" -d '{"query":"mutation { createEvent(name: \"demo\", when: \"Next Friday!\", where: \"Here!\") {id name when where description}}"}' https://<ENDPOINT>/graphql
```

Potrai quindi accedervi con `ctx.request.headers.custom`. Ad esempio, potrebbe essere nel codice seguente per DynamoDB:

```
"custom": util.dynamodb.toDynamoDB(ctx.request.headers.custom)
```

Esempio di intestazione multipla

Puoi anche passare più intestazioni in una singola richiesta e accedervi nel gestore del resolver. Ad esempio, se l'intestazione è impostata con due valori:

```
curl -XPOST -H "Content-Type:application/graphql" -H "custom:bailey" -H "custom:nadia" \
-H "x-api-key:<API-KEY-VALUE>" -d '{"query":"mutation { createEvent(name: \"demo \", when: \"Next Friday!\", where: \"Here!\") {id name when where description}}}' \
https://<ENDPOINT>/graphql
```

Potrai quindi accedervi come matrice, ad esempio `ctx.request.headers.custom[1]`.

Note

AWS AppSync non espone l'intestazione del cookie in `ctx.request.headers`

Accedi al nome di dominio personalizzato della richiesta

AWS AppSync supporta la configurazione di un dominio personalizzato che puoi utilizzare per accedere ai tuoi endpoint GraphQL e in tempo reale per il tuo. APIs Quando si effettua una richiesta con un nome di dominio personalizzato, è possibile ottenere il nome di dominio utilizzando.

`ctx.request.domainName`

Quando si utilizza il nome di dominio endpoint GraphQL predefinito, il valore è `null`

Informazioni

La sezione `info` contiene informazioni sulla richiesta GraphQL. Questa sezione ha il seguente formato:

```
type Info = {
  fieldName: string;
  parentTypeName: string;
  variables: any;
  selectionSetList: string[];
  selectionSetGraphQL: string;
};
```

Ogni campo è definito nel modo seguente:

fieldName

Il nome del campo attualmente in corso di risoluzione.

parentTypeName

Il nome del tipo padre per il campo attualmente in corso di risoluzione.

variables

Una mappa contenente tutte le variabili che vengono passate nella richiesta GraphQL.

selectionSetList

Rappresentazione elenco dei campi nel set di selezione GraphQL. I campi con alias sono referenziati solo dal nome dell'alias, non dal nome del campo. L'esempio seguente mostra questa indicazione in dettaglio.

selectionSetGraphQL

Rappresentazione stringa del set di selezione, formattata come SDL (Schema Definition Language) GraphQL. Sebbene i frammenti non vengano uniti nel set di selezione, i frammenti in linea vengono conservati, come illustrato nell'esempio seguente.

 Note

JSON.stringify non includerà selectionSetGraphQL e selectionSetList nella serializzazione delle stringhe. È necessario fare riferimento direttamente a queste proprietà.

Ad esempio, se stai risolvendo il campo `getPost` della query seguente:

```
query {  
  getPost(id: $postId) {  
    postId  
    title  
    secondTitle: title  
    content  
    author(id: $authorId) {  
      authorId  
      name  
    }  
  }  
}
```

```

    }
    secondAuthor(id: "789") {
      authorId
    }
    ... on Post {
      inlineFrag: comments: {
        id
      }
    }
    ... postFrag
  }
}

fragment postFrag on Post {
  postFrag: comments: {
    id
  }
}

```

Quindi la `ctx.info` variabile completa disponibile durante l'elaborazione di un gestore potrebbe essere:

```

{
  "fieldName": "getPost",
  "parentTypeName": "Query",
  "variables": {
    "postId": "123",
    "authorId": "456"
  },
  "selectionSetList": [
    "postId",
    "title",
    "secondTitle",
    "content",
    "author",
    "author/authorId",
    "author/name",
    "secondAuthor",
    "secondAuthor/authorId",
    "inlineFragComments",
    "inlineFragComments/id",
    "postFragComments",
    "postFragComments/id"
  ]
}

```

```

],
"selectionSetGraphQL": "{\n  getPost(id: $postId) {\n    postId\n    title\n    secondTitle: title\n    content\n    author(id: $authorId) {\n      authorId\n      name\n    }\n    secondAuthor(id: \"789\") {\n      authorId\n    }\n    ... on Post\n    {\n      inlineFrag: comments {\n        id\n      }\n      ... postFrag\n    }\n  }"
}

```

`selectionSetListesponse` solo i campi che appartengono al tipo corrente. Se il tipo corrente è un'interfaccia o un'unione, vengono esposti solo i campi selezionati che appartengono all'interfaccia. Ad esempio, dato lo schema seguente:

```

type Query {
  node(id: ID!): Node
}

interface Node {
  id: ID
}

type Post implements Node {
  id: ID
  title: String
  author: String
}

type Blog implements Node {
  id: ID
  title: String
  category: String
}

```

E la seguente domanda:

```

query {
  node(id: "post1") {
    id
    ... on Post {
      title
    }

    ... on Blog {
      title
    }
  }
}

```

```
    }  
  }  
}
```

Quando si chiama `ctx.info.selectionSetList` alla risoluzione del `Query.node` campo, `id` viene esposto solo:

```
"selectionSetList": [  
  "id"  
]
```

AWS AppSync JavaScript funzionalità di runtime per resolver e funzioni

L'ambiente `APPSYNC_JS` di runtime offre funzionalità simili alla versione 6.0 [ECMAScript \(ES\)](#). Supporta un sottoinsieme delle sue funzionalità e fornisce alcuni metodi aggiuntivi (utilità) che non fanno parte delle specifiche ES. Negli argomenti seguenti sono elencate tutte le funzionalità linguistiche supportate:

- [Funzionalità di runtime supportate](#): scopri di più sulle funzionalità di base supportate, sugli oggetti primitivi, sugli oggetti e sulle funzioni incorporati, ecc.
- [Utilità integrate](#): la variabile `util` contiene metodi di utilità generali per aiutarti a lavorare con i dati. Se non diversamente specificato, tutte le utilità usano il set di caratteri UTF-8.
- [Moduli integrati](#): scopri di più su come i moduli integrati possono aiutare a scrivere JavaScript resolver e funzioni.
- [Utilità di runtime](#): la libreria di runtime fornisce utilità per controllare o modificare le proprietà di runtime dei resolver e delle funzioni.
- [Time helpers in util.time - La variabile util.time](#) contiene metodi `datetime` per aiutare a generare timestamp, convertire tra formati `datetime` e analizzare stringhe `datetime`. Si basa sulla sintassi per i formati `datetime`, a cui è possibile fare riferimento per ulteriore documentazione. [DateTimeFormatter](#)
- [Helper DynamoDB in util.dynamodb - util.dynamodb contiene metodi di supporto che semplificano la scrittura e la lettura dei dati su Amazon DynamoDB, come la mappatura](#) e la formattazione automatiche dei tipi.
- [Helper HTTP in util.http - L'utilità util.http fornisce metodi di supporto che è possibile utilizzare per gestire i parametri di richiesta HTTP](#) e aggiungere intestazioni di risposta.

- [Aiutanti di trasformazione in util.transform - util.transform](#) contiene metodi di supporto che semplificano l'esecuzione di operazioni complesse sulle fonti di dati.
- [String helpers in util.str - util.str](#) contiene metodi per facilitare le operazioni più comuni sulle stringhe.
- [Estensioni](#): le estensioni contengono una serie di metodi per eseguire azioni aggiuntive all'interno dei resolver.
- [Gli helper XML in util.xml](#) - util.xml contengono metodi per facilitare la conversione di stringhe XML.

Note

Attualmente, questo riferimento si applica solo alla versione di runtime 1.0.0.

Funzionalità di runtime supportate

Le sezioni seguenti descrivono il set di funzionalità supportate del runtime APPSYNC_JS.

Caratteristiche principali

Sono supportate le seguenti funzionalità principali.

Types

Sono supportati i seguenti tipi:

- numeri
- stringhe
- booleani
- objects
- matrici
- funzioni

Operators

Gli operatori sono supportati, tra cui:

- operatori matematici standard (+, -, /, %, *, ecc.)

- operatore di coalescenza nullo (`() ??`)
- Concatenamento opzionale (`() ?.`)
- operatori bit per bit
- void operatori `typeof`
- operatori di diffusione (`...`)

I seguenti operatori non sono supportati:

- operatori unari (`++`, `--`, `e~`)
- Operatore `in`

 Note

Utilizzate `Object.hasOwnProperty` per verificare se la proprietà specificata si trova nell'oggetto specificato.

Statements

Sono supportate le seguenti istruzioni:

- `const`
- `let`
- `var`
- `break`
- `else`
- `for-in`
- `for-of`
- `if`
- `return`
- `switch`
- sintassi diffusa

Le seguenti non sono supportate:

- `catch`
- `continue`
- `do-while`
- `finally`
- `for(initialization; condition; afterthought)`

 Note

Le eccezioni sono `for-in` e `for-of` le espressioni, che sono supportate.

- `throw`
- `try`
- `while`
- dichiarazioni etichettate

Literals

Sono supportati i seguenti [valori letterali del modello](#) ES 6:

- Stringhe multilinea
- Interpolazione delle espressioni
- Modelli di nidificazione

Functions

È supportata la seguente sintassi della funzione:

- Le dichiarazioni di funzione sono supportate.
- Le funzioni a freccia ES 6 sono supportate.
- È supportata la sintassi dei parametri rest di ES 6.

Strict mode

Le funzioni funzionano in modalità rigorosa per impostazione predefinita, quindi non è necessario aggiungere una istruzione `use_strict` nel codice funzione. Non possono essere modificate.

Oggetti primitivi

Sono supportati i seguenti oggetti primitivi di ES e le relative funzioni.

Object

Sono supportati i seguenti oggetti:

- `Object.assign()`
- `Object.entries()`
- `Object.hasOwn()`
- `Object.keys()`
- `Object.values()`
- `delete`

String

Sono supportate le seguenti stringhe:

- `String.prototype.length()`
- `String.prototype.charAt()`
- `String.prototype.concat()`
- `String.prototype.endsWith()`
- `String.prototype.indexOf()`
- `String.prototype.lastIndexOf()`
- `String.raw()`
- `String.prototype.replace()`

Note

Le espressioni regolari non sono supportate.

Tuttavia, i costrutti di espressioni regolari in stile Java sono supportati nel parametro fornito. [Per ulteriori informazioni, vedete Pattern.](#)

- `String.prototype.replaceAll()`

 Note

Le espressioni regolari non sono supportate.

Tuttavia, i costrutti di espressioni regolari in stile Java sono supportati nel parametro fornito. [Per ulteriori informazioni, vedete Pattern.](#)

- `String.prototype.slice()`
- `String.prototype.split()`
- `String.prototype.startsWith()`
- `String.prototype.toLowerCase()`
- `String.prototype.toUpperCase()`
- `String.prototype.trim()`
- `String.prototype.trimEnd()`
- `String.prototype.trimStart()`

Number

Sono supportati i seguenti numeri:

- `Number.isFinite`
- `Number.isNaN`

Oggetti e funzioni incorporati

Sono supportate le funzioni e gli oggetti seguenti.

Math

Sono supportate le seguenti funzioni matematiche:

- `Math.random()`
- `Math.min()`
- `Math.max()`
- `Math.round()`

- `Math.floor()`
- `Math.ceil()`

Array

Sono supportati i seguenti metodi di matrice:

- `Array.prototype.length`
- `Array.prototype.concat()`
- `Array.prototype.fill()`
- `Array.prototype.flat()`
- `Array.prototype.indexOf()`
- `Array.prototype.join()`
- `Array.prototype.lastIndexOf()`
- `Array.prototype.pop()`
- `Array.prototype.push()`
- `Array.prototype.reverse()`
- `Array.prototype.shift()`
- `Array.prototype.slice()`
- `Array.prototype.sort()`

Note

`Array.prototype.sort()` non supporta argomenti.

- `Array.prototype.splice()`
- `Array.prototype.unshift()`
- `Array.prototype.forEach()`
- `Array.prototype.map()`
- `Array.prototype.flatMap()`
- `Array.prototype.filter()`
- `Array.prototype.reduce()`

- `Array.prototype.reduceRight()`
- `Array.prototype.find()`
- `Array.prototype.some()`
- `Array.prototype.every()`
- `Array.prototype.findIndex()`
- `Array.prototype.findLast()`
- `Array.prototype.findLastIndex()`
- `delete`

Console

L'oggetto console è disponibile per il debug. Durante l'esecuzione delle query in tempo reale, log/error le istruzioni della console vengono inviate ad Amazon CloudWatch Logs (se la registrazione è abilitata). Durante la valutazione del codice `conevaluateCode`, le istruzioni di log vengono restituite nella risposta al comando.

- `console.error()`
- `console.log()`

Function

- I call metodi `apply` e `bind`, e non sono supportati.
- I costruttori di funzioni non sono supportati.
- Il passaggio di una funzione come argomento non è supportato.
- Le chiamate di funzioni ricorsive non sono supportate.

JSON

Sono supportati i seguenti metodi JSON:

- `JSON.parse()`

Note

Restituisce una stringa vuota se la stringa analizzata non è un formato JSON valido.

- `JSON.stringify()`

Promises

I processi asincroni non sono supportati e le promesse non sono supportate.

Note

L'accesso alla rete e al file system non è supportato nel APPSYNC_JS runtime di AWS AppSync. AWS AppSync gestisce tutte le operazioni di I/O in base alle richieste effettuate dal AWS AppSync resolver o dalla funzione. AWS AppSync

Elementi globali

Sono supportate le seguenti costanti globali:

- `NaN`
- `Infinity`
- `undefined`
- [`util`](#)
- [`extensions`](#)
- [`runtime`](#)

Tipi di errore

Il lancio di errori con `non throw` è supportato. È possibile restituire un errore utilizzando `util.error()` la funzione. È possibile includere un errore nella risposta GraphQL utilizzando la `util.appendError` funzione.

Per ulteriori informazioni, consulta [Error utils](#).

Utilità integrate

La `util` variabile contiene metodi di utilità generali per aiutarvi a lavorare con i dati. Se non diversamente specificato, tutte le utilità usano il set di caratteri UTF-8.

Utilità di codifica

Elenco degli strumenti di codifica

util.urlEncode(String)

Restituisce la stringa di input come stringa codificata `application/x-www-form-urlencoded`.

util.urlDecode(String)

Decodifica una stringa codificata `application/x-www-form-urlencoded` nella relativa forma non codificata.

util.base64Encode(string) : string

Codifica l'input in una stringa con codifica base64.

util.base64Decode(string) : string

Decodifica i dati da una stringa con codifica base64.

Utilità per la generazione di ID

Elenco di utilità per la generazione di ID

util.autoId()

Restituisce un valore UUID generato casualmente a 128 bit.

util.autoUlid()

Restituisce un ULID (Universally Unique Lexicographically Sortable Identifier) generato casualmente a 128 bit.

util.autoKsuid()

Restituisce un KSUID (K-Sortable Unique Identifier) base62 generato casualmente a 128 bit codificato come String con una lunghezza di 27.

Utili di errore

Elenco delle utilità di errore

util.error(String, String?, Object?, Object?)

Genera un errore personalizzato. Può essere usato nei modelli di mappatura di richieste o risposte se il modello rileva un errore nella richiesta o nel risultato della chiamata. Inoltre, è `errorType` possibile specificare un `data` campo, un `errorInfo` campo e un campo. Il valore di `data` verrà aggiunto al blocco `error` corrispondente all'interno di `errors` nella risposta di GraphQL.

Note

`data` verrà filtrato in base al set di selezione dell'interrogazione. Il valore di `errorInfo` verrà aggiunto al blocco `error` corrispondente all'interno di `errors` nella risposta di GraphQL.

`errorInfo` non verrà filtrato in base al set di selezione delle interrogazioni.

util.appendError(String, String?, Object?, Object?)

Aggiunge un errore personalizzato. Può essere usato nei modelli di mappatura di richieste o risposte se il modello rileva un errore nella richiesta o nel risultato della chiamata. Inoltre, è `errorType` possibile specificare un `data` campo, un `errorInfo` campo e un campo. A differenza di `util.error(String, String?, Object?, Object?)`, la valutazione del modello non viene interrotta, in modo che i dati possano essere restituiti al chiamante. Il valore di `data` verrà aggiunto al blocco `error` corrispondente all'interno di `errors` nella risposta di GraphQL.

Note

`data` verrà filtrato in base al set di selezione dell'interrogazione. Il valore di `errorInfo` verrà aggiunto al blocco `error` corrispondente all'interno di `errors` nella risposta di GraphQL.

`errorInfo` non verrà filtrato in base al set di selezione delle interrogazioni.

Utilità per la corrispondenza di tipi e modelli

Elenco degli strumenti per la corrispondenza di tipi e modelli

util.matches(String, String) : Boolean

Restituisce true se il modello specificato nel primo argomento corrisponde ai dati forniti nel secondo argomento. Il modello deve essere un'espressione regolare, ad esempio `util.matches("a*b", "aaaaab")`. La funzionalità si basa sulla classe [Pattern](#) che puoi consultare per ottenere altre informazioni.

util.authType()

Restituisce una stringa che descrive il tipo di autenticazione multipla utilizzato da una richiesta, restituendo «IAM Authorization», «User Pool Authorization», «Open ID Connect Authorization» o «API Key Authorization».

Restituisce il valore di comportamento (utils)

Elenco delle utilità di comportamento del valore restituito

util.escapeJavaScript(String)

Restituisce la stringa di input come stringa di JavaScript escape.

Utilità di autorizzazione Resolver

Elenco degli strumenti di autorizzazione del resolver

util.unauthorized()

Genera Unauthorized per il campo in fase di risoluzione. Utilizzalo nei modelli di mappatura delle richieste o delle risposte per determinare se consentire al chiamante di risolvere il campo.

Moduli incorporati

I moduli fanno parte del APPSYNC_JS runtime e forniscono utilità per aiutare a scrivere JavaScript resolver e funzioni. Per esempi ed esempi, consultate il repository. [aws-appsync-resolver-samples](#) GitHub

Funzioni del modulo DynamoDB

Le funzioni del modulo DynamoDB offrono un'esperienza migliorata durante l'interazione con le fonti di dati DynamoDB. È possibile effettuare richieste verso le sorgenti dati DynamoDB utilizzando le funzioni e senza aggiungere la mappatura dei tipi.

I moduli vengono importati utilizzando: `@aws-appsync/utils/dynamodb`

```
// Modules are imported using @aws-appsync/utils/dynamodb
import * as ddb from '@aws-appsync/utils/dynamodb';
```

Funzioni

Elenco delle funzioni

get<T>(payload: GetInput): DynamoDBGetItemRequest

Tip

[the section called “Input”](#) Per informazioni su, vedere GetInput.

Genera un `DynamoDBGetItemRequest` oggetto per effettuare una [GetItem](#) richiesta a DynamoDB.

```
import { get } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  return get({ key: { id: ctx.args.id } });
}
```

put<T>(payload): DynamoDBPutItemRequest

Genera un `DynamoDBPutItemRequest` oggetto per effettuare una [PutItem](#) richiesta a DynamoDB.

```
import * as ddb from '@aws-appsync/utils/dynamodb'

export function request(ctx) {
  return ddb.put({ key: { id: util.autoId() }, item: ctx.args });
}
```

```
}
```

remove<T>(payload): DynamoDBDeleteItemRequest

Genera un `DynamoDBDeleteItemRequest` oggetto per effettuare una [DeleteItem](#) richiesta a DynamoDB.

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  return ddb.remove({ key: { id: ctx.args.id } });
}
```

scan<T>(payload): DynamoDBScanRequest

Genera una richiesta `DynamoDBScanRequest` di [scansione](#) per DynamoDB.

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  const { limit = 10, nextToken } = ctx.args;
  return ddb.scan({ limit, nextToken });
}
```

sync<T>(payload): DynamoDBSyncRequest

Genera un `DynamoDBSyncRequest` oggetto per effettuare una richiesta di [sincronizzazione](#). La richiesta riceve solo i dati modificati dall'ultima query (delta updates). Le richieste possono essere effettuate solo a sorgenti dati DynamoDB con versione.

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  const { limit = 10, nextToken, lastSync } = ctx.args;
  return ddb.sync({ limit, nextToken, lastSync });
}
```

update<T>(payload): DynamoDBUpdateItemRequest

Genera un `DynamoDBUpdateItemRequest` oggetto per effettuare una [UpdateItem](#) richiesta a DynamoDB.

Operazioni

Gli assistenti operativi consentono di eseguire azioni specifiche su parti dei dati durante gli aggiornamenti. Per iniziare, importa `operations` da `@aws-appsync/utils/dynamodb`:

```
// Modules are imported using operations
import {operations} from '@aws-appsync/utils/dynamodb';
```

Elenco delle operazioni

add<T>(payload)

Una funzione di supporto che aggiunge un nuovo elemento di attributo durante l'aggiornamento di DynamoDB.

Esempio

Per aggiungere un indirizzo (via, città e codice postale) a un elemento DynamoDB esistente utilizzando il valore ID:

```
import { update, operations } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  const updateObj = {
    address: operations.add({
      street1: '123 Main St',
      city: 'New York',
      zip: '10001',
    }),
  };
  return update({ key: { id: 1 }, update: updateObj });
}
```

append <T>(payload)

Una funzione di supporto che aggiunge un payload all'elenco esistente in DynamoDB.

Esempio

Per aggiungere friend IDs (`newFriendIds`) appena aggiunto a una lista di amici esistente (`friendsIds`) durante un aggiornamento:

```
import { update, operations } from '@aws-appsync/utils/dynamodb';
```

```
export function request(ctx) {
  const newFriendIds = [101, 104, 111];
  const updateObj = {
    friendsIds: operations.append(newFriendIds),
  };
  return update({ key: { id: 1 }, update: updateObj });
}
```

decrement (by?)

Una funzione di supporto che decrementa il valore dell'attributo esistente nell'elemento durante l'aggiornamento di DynamoDB.

Esempio

Per diminuire di 10 il contatore di un amico (): friendsCount

```
import { update, operations } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  const updateObj = {
    friendsCount: operations.decrement(10),
  };
  return update({ key: { id: 1 }, update: updateObj });
}
```

increment (by?)

Una funzione di supporto che incrementa il valore dell'attributo esistente nell'elemento durante l'aggiornamento di DynamoDB.

Esempio

Per incrementare il contatore di amici () di 10: friendsCount

```
import { update, operations } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  const updateObj = {
    friendsCount: operations.increment(10),
  };
  return update({ key: { id: 1 }, update: updateObj });
}
```

```
}
```

prepend <T>(payload)

Una funzione di supporto che si aggiunge all'elenco esistente in DynamoDB.

Esempio

Per aggiungere friend IDs (`newFriendIds`) appena aggiunto a un elenco di amici esistente (`friendsIds`) durante un aggiornamento:

```
import { update, operations } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  const newFriendIds = [101, 104, 111];
  const updateObj = {
    friendsIds: operations.prepend(newFriendIds),
  };
  return update({ key: { id: 1 }, update: updateObj });
}
```

replace <T>(payload)

Una funzione di supporto che sostituisce un attributo esistente durante l'aggiornamento di un elemento in DynamoDB. È utile quando si desidera aggiornare l'intero oggetto o sottooggetto nell'attributo e non solo le chiavi nel payload.

Esempio

Per sostituire un indirizzo (via, città e codice postale) in un oggetto: `info`

```
import { update, operations } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  const updateObj = {
    info: {
      address: operations.replace({
        street1: '123 Main St',
        city: 'New York',
        zip: '10001',
      }),
    },
  };
}
```

```
return update({ key: { id: 1 }, update: updateObj });
}
```

updateListItem <T>(payload, index)

Una funzione di supporto che sostituisce un elemento in un elenco.

Esempio

Nell'ambito di `update (newFriendIds)`, questo esempio veniva utilizzato `updateListItem` per aggiornare i valori ID del secondo elemento (`index:1`, new ID:102) e del terzo elemento (`index:2`, new ID:112) in un elenco (`friendsIds`).

```
import { update, operations as ops } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  const newFriendIds = [
    ops.updateListItem('102', 1), ops.updateListItem('112', 2)
  ];
  const updateObj = { friendsIds: newFriendIds };
  return update({ key: { id: 1 }, update: updateObj });
}
```

Input

Elenco degli input

Type GetInput<T>

```
GetInput<T>: {
  consistentRead?: boolean;
  key: DynamoDBKey<T>;
}
```

Dichiarazione del tipo

- `consistentRead?: boolean` (facoltativo)

Un booleano opzionale per specificare se si desidera eseguire una lettura fortemente coerente con DynamoDB.

- `key: DynamoDBKey<T>` (obbligatorio)

Un parametro obbligatorio che specifica la chiave dell'elemento in DynamoDB. Gli elementi DynamoDB possono avere una sola chiave hash o chiavi di hash e ordinamento.

Type **PutInput<T>**

```
PutInput<T>: {  
  _version?: number;  
  condition?: DynamoDBFilterObject<T> | null;  
  customPartitionKey?: string;  
  item: Partial<T>;  
  key: DynamoDBKey<T>;  
  populateIndexFields?: boolean;  
}
```

Dichiarazione del tipo

- `_version?: number` (facoltativo)
- `condition?: DynamoDBFilterObject<T> | null` (facoltativo)

Quando si inserisce un oggetto in una tabella DynamoDB, è possibile specificare facoltativamente un'espressione condizionale che controlli se la richiesta deve avere esito positivo o meno in base allo stato dell'oggetto già in DynamoDB prima dell'esecuzione dell'operazione.

- `customPartitionKey?: string` (facoltativo)

Se abilitato, questo valore di stringa modifica il formato dei `ds_pk` record `ds_sk` and utilizzati dalla tabella delta sync quando il controllo delle versioni è abilitato. Se abilitata, è abilitata anche l'elaborazione della `populateIndexFields` voce.

- `item: Partial<T>`(richiesto)

Il resto degli attributi dell'elemento da inserire in DynamoDB.

- `key: DynamoDBKey<T>`(richiesto)

Un parametro obbligatorio che specifica la chiave dell'elemento in DynamoDB su cui verrà eseguito il put. Gli elementi DynamoDB possono avere una sola chiave hash o chiavi di hash e ordinamento.

- `populateIndexFields?: boolean` (facoltativo)

Un valore booleano che, se abilitato insieme a `customPartitionKey`, crea nuove voci per ogni record nella tabella delta sync, in particolare nelle colonne `and.gsi_ds_pk` `gsi_ds_sk`. Per ulteriori informazioni, consulta [Rilevamento e sincronizzazione dei conflitti](#) nella Guida per gli AWS AppSync sviluppatori.

Type `QueryInput<T>`

```
QueryInput<T>: ScanInput<T> & {
  query: DynamoDBKeyCondition<Required<T>>;
}
```

Dichiarazione del tipo

- `query: DynamoDBKeyCondition<Required<T>>`(richiesto)

Specifica una condizione chiave che descrive gli elementi da interrogare. Per un determinato indice, la condizione per una chiave di partizione deve essere un'uguaglianza e la chiave di ordinamento un confronto o un `beginsWith` (quando è una stringa). Per le chiavi di partizione e ordinamento sono supportati solo i tipi di numeri e stringhe.

Esempio

Prendi il `User` tipo seguente:

```
type User = {
  id: string;
  name: string;
  age: number;
  isVerified: boolean;
  friendsIds: string[]
}
```

La query può includere solo i seguenti campi: `id` `name`, `age`:

```
const query: QueryInput<User> = {
  name: { eq: 'John' },
  age: { gt: 20 },
}
```

Type RemoveInput<T>

```
RemoveInput<T>: {
  _version?: number;
  condition?: DynamoDBFilterObject<T>;
  customPartitionKey?: string;
  key: DynamoDBKey<T>;
  populateIndexFields?: boolean;
}
```

Dichiarazione di tipo

- `_version?: number` (facoltativo)
- `condition?: DynamoDBFilterObject<T>` (facoltativo)

Quando rimuovi un oggetto in DynamoDB, puoi facoltativamente specificare un'espressione condizionale che controlli se la richiesta deve avere successo o meno in base allo stato dell'oggetto già in DynamoDB prima dell'esecuzione dell'operazione.

Esempio

L'esempio seguente è un `DeleteItem` espressione contenente una condizione che consente l'esito positivo dell'operazione solo se il proprietario del documento corrisponde all'utente che effettua la richiesta.

```
type Task = {
  id: string;
  title: string;
  description: string;
  owner: string;
  isComplete: boolean;
}
const condition: DynamoDBFilterObject<Task> = {
  owner: { eq: 'XXXXXXXXXXXXXXXXXX' },
}

remove<Task>({
  key: {
    id: 'XXXXXXXXXXXXXXXXXX',
  },
  condition,
});
```

- `customPartitionKey?: string` (facoltativo)

Se abilitato, il `customPartitionKey` valore modifica il formato dei `ds_pk` record `ds_sk` and utilizzati dalla tabella delta sync quando il controllo delle versioni è abilitato. Se abilitata, è abilitata anche l'elaborazione della `populateIndexFields` voce.

- `key: DynamoDBKey<T>` (richiesto)

Un parametro obbligatorio che specifica la chiave dell'elemento in DynamoDB che viene rimosso. Gli elementi DynamoDB possono avere una sola chiave hash o chiavi di hash e ordinamento.

Esempio

Se a ha la chiave hash `User` solo con un `utentelid`, la chiave sarebbe simile a questa:

```
type User = {  
  id: number  
  name: string  
  age: number  
  isVerified: boolean  
}  
const key: DynamoDBKey<User> = {  
  id: 1,  
}
```

Se l'utente della tabella ha una chiave hash (`id`) e una chiave di ordinamento (`name`), la chiave sarebbe simile a questa:

```
type User = {  
  id: number  
  name: string  
  age: number  
  isVerified: boolean  
  friendsIds: string[]  
}  
  
const key: DynamoDBKey<User> = {  
  id: 1,  
  name: 'XXXXXXXXXX',  
}
```

- `populateIndexFields?: boolean` (facoltativo)

Un valore booleano che, se abilitato insieme a `customPartitionKey`, crea nuove voci per ogni record nella tabella delta sync, in particolare nelle `gsi_ds_pk` colonne and. `gsi_ds_sk`

Type `ScanInput<T>`

```
ScanInput<T>: {  
  consistentRead?: boolean | null;  
  filter?: DynamoDBFilterObject<T> | null;  
  index?: string | null;  
  limit?: number | null;  
  nextToken?: string | null;  
  scanIndexForward?: boolean | null;  
  segment?: number;  
  select?: DynamoDBSelectAttributes;  
  totalSegments?: number;  
}
```

Dichiarazione del tipo

- `consistentRead?: boolean | null` (facoltativo)

Un booleano opzionale per indicare letture coerenti quando si esegue una query su DynamoDB. Il valore predefinito è `false`.

- `filter?: DynamoDBFilterObject<T> | null` (facoltativo)

Un filtro opzionale da applicare ai risultati dopo averlo recuperato dalla tabella.

- `index?: string | null` (facoltativo)

Un nome opzionale dell'indice da scansionare.

- `limit?: number | null` (facoltativo)

Un numero massimo opzionale di risultati da restituire.

- `nextToken?: string | null` (facoltativo)

Un token di impaginazione opzionale per continuare una query precedente. dalla quale deve essere ottenuto.

- `scanIndexForward?: boolean | null` (facoltativo)

Un booleano opzionale per indicare se la query viene eseguita in ordine crescente o decrescente. Per impostazione predefinita, questo valore è impostato su `true`.

- `segment?: number` (facoltativo)
- `select?: DynamoDBSelectAttributes` (facoltativo)

Attributi da restituire da DynamoDB. Per impostazione predefinita, il AWS AppSync resolver DynamoDB restituisce solo gli attributi proiettati nell'indice. I valori supportati sono:

- `ALL_ATTRIBUTES`

Restituisce tutti gli attributi degli elementi dalla tabella o dall'indice specificato. Se si esegue una query su un indice secondario locale, DynamoDB recupera l'intero elemento dalla tabella principale per ogni elemento corrispondente nell'indice. Se l'indice è configurato per proiettare tutti gli attributi della voce, è possibile ottenere tutti i dati dall'indice secondario locale, senza necessità di recupero.

- `ALL_PROJECTED_ATTRIBUTES`

Restituisce tutti gli attributi che sono stati proiettati nell'indice. Se la configurazione dell'indice prevede che vi siano proiettati tutti gli attributi, il valore che restituisce è uguale a quello dato da `ALL_ATTRIBUTES`.

- `SPECIFIC_ATTRIBUTES`

Restituisce solo gli attributi elencati in `ProjectionExpression`. Questo valore restituito equivale a specificare `ProjectionExpression` senza specificare alcun valore per `AttributesToGet`

- `totalSegments?: number` (facoltativo)

Type `DynamoDBSyncInput<T>`

```
DynamoDBSyncInput<T>: {  
  basePartitionKey?: string;  
  deltaIndexName?: string;  
  filter?: DynamoDBFilterObject<T> | null;  
  lastSync?: number;  
  limit?: number | null;  
  nextToken?: string | null;  
}
```

Dichiarazione del tipo

- `basePartitionKey?: string` (facoltativo)

La chiave di partizione della tabella di base da utilizzare durante l'esecuzione di un'operazione di sincronizzazione. Questo campo consente di eseguire un'operazione di sincronizzazione quando la tabella utilizza una chiave di partizione personalizzata.

- `deltaIndexName?: string` (facoltativo)

L'indice utilizzato per l'operazione di sincronizzazione. Questo indice è necessario per abilitare un'operazione di sincronizzazione sull'intera tabella delta store quando la tabella utilizza una chiave di partizione personalizzata. L'operazione di sincronizzazione verrà eseguita sul GSI (creato su `gsi_ds_pk` and `gsi_ds_sk`).

- `filter?: DynamoDBFilterObject<T> | null` (facoltativo)

Un filtro opzionale da applicare ai risultati dopo averlo recuperato dalla tabella.

- `lastSync?: number` (facoltativo)

Il momento, in millisecondi di epoca, in cui è iniziata l'ultima operazione di sincronizzazione riuscita. Se specificato, vengono restituiti solo gli elementi che sono stati modificati dopo `lastSync`. Questo campo deve essere compilato solo dopo aver recuperato tutte le pagine da un'operazione di sincronizzazione iniziale. Se omesso, verranno restituiti i risultati della tabella di base. In caso contrario, verranno restituiti i risultati della tabella delta.

- `limit?: number | null` (facoltativo)

Un numero massimo opzionale di elementi da valutare contemporaneamente. Se omesso, il limite predefinito sarà impostato su 100 elementi. Il valore massimo per questo campo è 1000 articoli.

- `nextToken?: string | null` (facoltativo)

Type `DynamoDBUpdateInput<T>`

```
DynamoDBUpdateInput<T>: {  
  _version?: number;  
  condition?: DynamoDBFilterObject<T>;  
  customPartitionKey?: string;  
  key: DynamoDBKey<T>;  
  populateIndexFields?: boolean;  
  update: DynamoDBUpdateObject<T>;  
}
```

Tipo di dichiarazione

- `_version?: number` (facoltativo)
- `condition?: DynamoDBFilterObject<T>` (facoltativo)

Quando aggiorni un oggetto in DynamoDB, puoi facoltativamente specificare un'espressione condizionale che controlli se la richiesta deve avere successo o meno in base allo stato dell'oggetto già in DynamoDB prima dell'esecuzione dell'operazione.

- `customPartitionKey?: string` (facoltativo)

Se abilitato, il `customPartitionKey` valore modifica il formato dei `ds_pk` record `ds_sk` and utilizzati dalla tabella delta sync quando il controllo delle versioni è abilitato. Se abilitata, è abilitata anche l'elaborazione della `populateIndexFields` voce.

- `key: DynamoDBKey<T>`(richiesto)

Un parametro obbligatorio che specifica la chiave dell'elemento in DynamoDB che viene aggiornato. Gli elementi DynamoDB possono avere una sola chiave hash o chiavi di hash e ordinamento.

- `populateIndexFields?: boolean` (facoltativo)

Un valore booleano che, se abilitato insieme a `customPartitionKey`, crea nuove voci per ogni record nella tabella delta sync, in particolare nelle colonne `and.gsi_ds_pk` `gsi_ds_sk`

- `update: DynamoDBUpdateObject<T>`

Un oggetto che specifica gli attributi da aggiornare insieme ai relativi nuovi valori. L'oggetto di aggiornamento può essere utilizzato con `add`, `remove`, `replace`, `increment`, `decrement`, `append`, `prepend`, `updateListItem`.

Funzioni del modulo Amazon RDS

Le funzioni del modulo Amazon RDS offrono un'esperienza migliorata durante l'interazione con i database configurati con l'API Amazon RDS Data. Il modulo viene importato utilizzando: `@aws-appsync/utils/rds`

```
import * as rds from '@aws-appsync/utils/rds';
```

Le funzioni possono essere importate anche singolarmente. Ad esempio, l'importazione seguente utilizza `sql`:


```
import { sql } from '@aws-appsync/utils/rds';
```

Funzioni

È possibile utilizzare gli assistenti di utilità del modulo AWS AppSync RDS per interagire con il database.

Select

L'`select` utilità crea un'`SELECT` istruzione per interrogare il database relazionale.

Uso di base

Nella sua forma base, puoi specificare la tabella su cui vuoi interrogare:

```
import { select, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {

  // Generates statement:
  // "SELECT * FROM "persons"
  return createPgStatement(select({table: 'persons'}));
}
```

Nota che puoi anche specificare lo schema nell'identificatore della tabella:

```
import { select, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {

  // Generates statement:
  // SELECT * FROM "private"."persons"
  return createPgStatement(select({table: 'private.persons'}));
}
```

Specificare le colonne

È possibile specificare le colonne con la `columns` proprietà. Se non è impostato su un valore, il valore predefinito è: `*`

```
export function request(ctx) {
```

```
// Generates statement:
// SELECT "id", "name"
// FROM "persons"
return createPgStatement(select({
  table: 'persons',
  columns: ['id', 'name']
}));
}
```

Puoi anche specificare la tabella di una colonna:

```
export function request(ctx) {

  // Generates statement:
  // SELECT "id", "persons"."name"
  // FROM "persons"
  return createPgStatement(select({
    table: 'persons',
    columns: ['id', 'persons.name']
  }));
}
```

Limiti e offset

È possibile applicare limit e offset alla query:

```
export function request(ctx) {

  // Generates statement:
  // SELECT "id", "name"
  // FROM "persons"
  // LIMIT :limit
  // OFFSET :offset
  return createPgStatement(select({
    table: 'persons',
    columns: ['id', 'name'],
    limit: 10,
    offset: 40
  }));
}
```

Ordina per

Puoi ordinare i risultati in base alla `orderBy` proprietà. Fornisci una matrice di oggetti che specificano la colonna e una `dir` proprietà opzionale:

```
export function request(ctx) {

  // Generates statement:
  // SELECT "id", "name" FROM "persons"
  // ORDER BY "name", "id" DESC
  return createPgStatement(select({
    table: 'persons',
    columns: ['id', 'name'],
    orderBy: [{column: 'name'}, {column: 'id', dir: 'DESC'}]
  }));
}
```

Filtri

È possibile creare filtri utilizzando l'oggetto di condizione speciale:

```
export function request(ctx) {

  // Generates statement:
  // SELECT "id", "name"
  // FROM "persons"
  // WHERE "name" = :NAME
  return createPgStatement(select({
    table: 'persons',
    columns: ['id', 'name'],
    where: {name: {eq: 'Stephane'}}
  }));
}
```

Puoi anche combinare filtri:

```
export function request(ctx) {

  // Generates statement:
  // SELECT "id", "name"
  // FROM "persons"
  // WHERE "name" = :NAME and "id" > :ID
  return createPgStatement(select({
    table: 'persons',
```

```

        columns: ['id', 'name'],
        where: {name: {eq: 'Stephane'}, id: {gt: 10}}
    }));
}

```

Puoi anche creare OR dichiarazioni:

```

export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    // WHERE "name" = :NAME OR "id" > :ID
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        where: { or: [
            { name: { eq: 'Stephane' } },
            { id: { gt: 10 } }
        ]}
    }));
}

```

Puoi anche annullare una condizione connot:

```

export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    // WHERE NOT ("name" = :NAME AND "id" > :ID)
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        where: { not: [
            { name: { eq: 'Stephane' } },
            { id: { gt: 10 } }
        ]}
    }));
}

```

È inoltre possibile utilizzare i seguenti operatori per confrontare i valori:

Operatore	Description	Tipi di valori possibili
eq	Uguale	numero, stringa, booleano
uno	Non uguale	numero, stringa, booleano
le	Minore di o uguale a	numero, stringa
lt	Less than	numero, stringa
età	Maggiore di o uguale a	numero, stringa
gt	Greater than	numero, stringa
contiene	Piace	stringa
Non contiene	Non come	stringa
Inizia con	Inizia con il prefisso	stringa
tra	Tra due valori	numero, stringa
L'attributo esiste	L'attributo non è nullo	numero, stringa, booleano
formato	controlla la lunghezza dell'elemento	stringa

Insert

L'insertutilità fornisce un modo semplice per inserire elementi a riga singola nel database con l'operazione. INSERT

Inserimenti di singoli elementi

Per inserire un elemento, specifica la tabella e poi inserisci i valori dell'oggetto. Le chiavi degli oggetti vengono mappate alle colonne della tabella. I nomi delle colonne vengono espulsi automaticamente e i valori vengono inviati al database utilizzando la mappa variabile:

```
import { insert, createMySQLStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
```

```

const { input: values } = ctx.args;
const insertStatement = insert({ table: 'persons', values });

// Generates statement:
// INSERT INTO `persons`(`name`)
// VALUES(:NAME)
return createMySQLStatement(insertStatement)
}

```

Caso d'uso MySQL

Puoi combinare un insert seguito da a per select recuperare la riga inserita:

```

import { insert, select, createMySQLStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
  const { input: values } = ctx.args;
  const insertStatement = insert({ table: 'persons', values });
  const selectStatement = select({
    table: 'persons',
    columns: '*',
    where: { id: { eq: values.id } },
    limit: 1,
  });

  // Generates statement:
  // INSERT INTO `persons`(`name`)
  // VALUES(:NAME)
  // and
  // SELECT *
  // FROM `persons`
  // WHERE `id` = :ID
  return createMySQLStatement(insertStatement, selectStatement)
}

```

Caso d'uso Postgres

Con Postgres, è possibile utilizzare [returning](#) per ottenere dati dalla riga inserita. Accetta * o una matrice di nomi di colonne:

```

import { insert, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {

```

```

const { input: values } = ctx.args;
const insertStatement = insert({
  table: 'persons',
  values,
  returning: '*'
});

// Generates statement:
// INSERT INTO "persons"("name")
// VALUES(:NAME)
// RETURNING *
return createPgStatement(insertStatement)
}

```

Aggiorna

L'updateutilità consente di aggiornare le righe esistenti. È possibile utilizzare l'oggetto condizione per applicare modifiche alle colonne specificate in tutte le righe che soddisfano la condizione. Ad esempio, supponiamo di avere uno schema che ci consente di effettuare questa mutazione. Vogliamo aggiornare the name of Person con il id valore di3, ma solo se li conosciamo (known_since) dall'anno2000:

```

mutation Update {
  updatePerson(
    input: {id: 3, name: "Jon"},
    condition: {known_since: {ge: "2000"}}
  ) {
    id
    name
  }
}

```

Il nostro risolutore di aggiornamenti ha il seguente aspetto:

```

import { update, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
  const { input: { id, ...values }, condition } = ctx.args;
  const where = {
    ...condition,
    id: { eq: id },
  };
};

```

```

const updateStatement = update({
  table: 'persons',
  values,
  where,
  returning: ['id', 'name'],
});

// Generates statement:
// UPDATE "persons"
// SET "name" = :NAME, "birthday" = :BDAY, "country" = :COUNTRY
// WHERE "id" = :ID
// RETURNING "id", "name"
return createPgStatement(updateStatement)
}

```

Possiamo aggiungere un segno di spunta alla nostra condizione per assicurarci che venga aggiornata solo la riga con la chiave primaria `id` uguale a `3`. Allo stesso modo, per `Postgresinserts`, è possibile utilizzare `returning` per restituire i dati modificati.

Rimuovi

L'`remove` utilità consente di eliminare le righe esistenti. È possibile utilizzare l'oggetto condizione su tutte le righe che soddisfano la condizione. Nota che `delete` è una parola chiave riservata in JavaScript. `removed` dovrebbe essere usato invece:

```

import { remove, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
  const { input: { id }, condition } = ctx.args;
  const where = { ...condition, id: { eq: id } };
  const deleteStatement = remove({
    table: 'persons',
    where,
    returning: ['id', 'name'],
  });

  // Generates statement:
  // DELETE "persons"
  // WHERE "id" = :ID
  // RETURNING "id", "name"
  return createPgStatement(deleteStatement)
}

```


Casting

In alcuni casi, potresti volere una maggiore specificità sul tipo di oggetto corretto da utilizzare nella tua dichiarazione. È possibile utilizzare i suggerimenti di tipo forniti per specificare il tipo di parametri. AWS AppSync supporta lo [stesso tipo di suggerimenti](#) della Data API. Puoi trasmettere i tuoi parametri utilizzando le typeHint funzioni del AWS AppSync `rds` modulo.

L'esempio seguente consente di inviare un array come valore che viene trasmesso come oggetto JSON. Utilizziamo l'->operatore per recuperare l'elemento index 2 nell'array JSON:

```
import { sql, createPgStatement, toJsonObject, typeHint } from '@aws-appsync/utils/rds';

export function request(ctx) {
  const arr = ctx.args.list_of_ids
  const statement = sql`select ${typeHint.JSON(arr)}->2 as value`
  return createPgStatement(statement)
}

export function response(ctx) {
  return toJsonObject(ctx.result)[0][0].value
}
```

Il casting è utile anche per la gestione e il confronto DATE e TIME: TIMESTAMP

```
import { select, createPgStatement, typeHint } from '@aws-appsync/utils/rds';

export function request(ctx) {
  const when = ctx.args.when
  const statement = select({
    table: 'persons',
    where: { createdAt : { gt: typeHint.DATETIME(when) } }
  })
  return createPgStatement(statement)
}
```

Ecco un altro esempio che mostra come inviare la data e l'ora correnti:

```
import { sql, createPgStatement, typeHint } from '@aws-appsync/utils/rds';

export function request(ctx) {
```

```
const now = util.time.nowFormatted('YYYY-MM-dd HH:mm:ss')
return createPgStatement(sql`select ${typeHint.TIMESTAMP(now)}`)
}
```

Suggerimenti di tipo disponibili

- `typeHint.DATE`- Il parametro corrispondente viene inviato come oggetto del DATE tipo al database. Il formato accettato è YYYY-MM-DD.
- `typeHint.DECIMAL`- Il parametro corrispondente viene inviato come oggetto del DECIMAL tipo al database.
- `typeHint.JSON`- Il parametro corrispondente viene inviato come oggetto del JSON tipo al database.
- `typeHint.TIME`- Il valore del parametro stringa corrispondente viene inviato come oggetto del TIME tipo al database. Il formato accettato è HH:MM:SS[.FFF].
- `typeHint.TIMESTAMP`- Il valore del parametro stringa corrispondente viene inviato come oggetto del TIMESTAMP tipo al database. Il formato accettato è YYYY-MM-DD HH:MM:SS[.FFF].
- `typeHint.UUID`- Il valore del parametro stringa corrispondente viene inviato come oggetto del UUID tipo al database.

- Utilità di runtime

La `runtime` libreria fornisce utilità per controllare o modificare le proprietà di runtime dei resolver e delle funzioni.

Elenco delle utilità di runtime

`runtime.earlyReturn(obj?: unknown, returnOptions?: {skipTo: 'END' | 'NEXT'}): never`

L'invocazione di questa funzione interromperà l'esecuzione del gestore, della AWS AppSync funzione o del resolver corrente (Unit o Pipeline Resolver) a seconda del contesto corrente. L'oggetto specificato viene restituito come risultato.

- Quando viene chiamato in un gestore di richieste di AWS AppSync funzioni, l'origine dati e il gestore di risposta vengono ignorati e viene chiamato il gestore della richiesta di funzione successivo (o il gestore di risposte del resolver della pipeline se questa era l'ultima funzione).
- AWS AppSync

- Quando viene chiamato in un gestore di richieste del resolver AWS AppSync della pipeline, l'esecuzione della pipeline viene saltata e il gestore di risposte del resolver della pipeline viene chiamato immediatamente.
- Quando `returnOptions` viene fornito con `skipTo` impostato su «END», l'esecuzione della pipeline viene saltata e il gestore di risposta del resolver della pipeline viene chiamato immediatamente.
- Quando `returnOptions` viene fornito con `skipTo` impostato su «NEXT», l'esecuzione della funzione viene saltata e viene chiamato il successivo gestore di pipeline.

Esempio

```
import { runtime } from '@aws-appsync/utils'

export function request(ctx) {
  runtime.earlyReturn({ hello: 'world' })
  // code below is not executed
  return ctx.args
}

// never called because request returned early
export function response(ctx) {
  return ctx.result
}
```

Aiutanti temporali in `util.time`

La variabile `util.time` contiene metodi `datetime` che consentono di generare timestamp, convertire i diversi formati `datetime` e analizzare le stringhe `datetime`. La sintassi per i formati `datetime` si basa sulla [DateTimeFormatter](#) quale è possibile fare riferimento per ulteriore documentazione.

Elenco delle utilità temporali

`util.time.nowISO8601()`

[Restituisce una rappresentazione String di UTC in ISO8601 formato.](#)

`util.time.nowEpochSeconds()`

Restituisce il numero di secondi dall'epoca (Unix epoch) 1970-01-01T00:00:00Z a ora.

util.time.nowEpochMilliseconds()

Restituisce il numero di millisecondi dall'epoca (Unix epoch) 1970-01-01T00:00:00Z a ora.

util.time.nowFormatted(String)

Restituisce una stringa del timestamp corrente in UTC utilizzando il formato specificato da un tipo di input stringa.

util.time.nowFormatted(String, String)

Restituisce una stringa del timestamp corrente per un fuso orario utilizzando il formato e il fuso orario specificati da tipi di input stringa.

util.time.parseFormattedToEpochMilliseconds(String, String)

Analizza un timestamp passato come String insieme a un formato contenente un fuso orario, quindi restituisce il timestamp in millisecondi dall'epoca.

util.time.parseFormattedToEpochMilliseconds(String, String, String)

Analizza un timestamp passato come String insieme a un formato e a un fuso orario, quindi restituisce il timestamp in millisecondi dall'epoca.

util.time.parseISO8601ToEpochMilliseconds(String)

Analizza un ISO8601 timestamp passato come String, quindi restituisce il timestamp in millisecondi trascorsi da epoch.

util.time.epochMillisecondsToSeconds(long)

Converte un timestamp in formato epoca (Unix epoch) espresso in millisecondi in un timestamp in formato epoca (Unix epoch) espresso in secondi.

util.time.epochMillisecondsToISO8601(long)

Converte il timestamp di un'epoca in millisecondi in un timestamp. ISO8601

util.time.epochMillisecondsToFormatted(long, String)

Converte il timestamp di un'epoca in millisecondi, passato così a lungo, in un timestamp formattato secondo il formato fornito in UTC.

util.time.epochMillisecondsToFormatted(long, String, String)

Converte il timestamp di un'epoca in millisecondi, passato come long, in un timestamp formattato secondo il formato fornito nel fuso orario fornito.

Aiutanti DynamoDB in `util.dynamodb`

`util.dynamodb` contiene metodi di supporto che semplificano la scrittura e la lettura dei dati su Amazon DynamoDB, come la mappatura e la formattazione automatiche dei tipi.

A DynamoDB

Elenco di utilità `ToDynamoDB`

`util.dynamodb.toDynamoDB(Object)`

Strumento generale di conversione degli oggetti per DynamoDB che converte gli oggetti di input nella rappresentazione DynamoDB appropriata. Rappresenta alcuni tipi in un determinato modo. Ad esempio, usa elenchi ("L") invece di set ("SS", "NS", "BS"). Questo restituisce un oggetto che descrive il valore dell'attributo DynamoDB.

Esempio di stringa

```
Input:      util.dynamodb.toDynamoDB("foo")
Output:     { "S" : "foo" }
```

Esempio di numero

```
Input:      util.dynamodb.toDynamoDB(12345)
Output:     { "N" : 12345 }
```

Esempio booleano

```
Input:      util.dynamodb.toDynamoDB(true)
Output:     { "BOOL" : true }
```

Esempio di elenco

```
Input:      util.dynamodb.toDynamoDB([ "foo", 123, { "bar" : "baz" } ])
Output:     {
      "L" : [
        { "S" : "foo" },
        { "N" : 123 },
        {
          "M" : {
            "bar" : { "S" : "baz" }
          }
        }
      ]
    }
```

```

    }
  }
}

```

Esempio di mappa

```

Input:      util.dynamodb.toDynamoDB({ "foo": "bar", "baz" : 1234, "beep":
[ "boop" ] })
Output:     {
    "M" : {
      "foo" : { "S" : "bar" },
      "baz" : { "N" : 1234 },
      "beep" : {
        "L" : [
          { "S" : "boop" }
        ]
      }
    }
  }
}

```

Utilità ToString

Elenco di utilità ToString

util.dynamodb.toString(String)

Converte una stringa di input nel formato stringa DynamoDB. Questo restituisce un oggetto che descrive il valore dell'attributo DynamoDB.

```

Input:      util.dynamodb.toString("foo")
Output:     { "S" : "foo" }

```

util.dynamodb.toStringSet(List<String>)

Converte un elenco con stringhe nel formato del set di stringhe DynamoDB. Questo restituisce un oggetto che descrive il valore dell'attributo DynamoDB.

```

Input:      util.dynamodb.toStringSet([ "foo", "bar", "baz" ])
Output:     { "SS" : [ "foo", "bar", "baz" ] }

```

ToNumber utils

Elenco di utilità ToNumber

util.dynamodb.toNumber(Number)

Converte un numero nel formato numerico DynamoDB. Questo restituisce un oggetto che descrive il valore dell'attributo DynamoDB.

```
Input:      util.dynamodb.toNumber(12345)
Output:     { "N" : 12345 }
```

util.dynamodb.toNumberSet(List<Number>)

Converte un elenco di numeri nel formato del set di numeri DynamoDB. Questo restituisce un oggetto che descrive il valore dell'attributo DynamoDB.

```
Input:      util.dynamodb.toNumberSet([ 1, 23, 4.56 ])
Output:     { "NS" : [ 1, 23, 4.56 ] }
```

ToBinary utils

Elenco di utilità ToBinary

util.dynamodb.toBinary(String)

Converte i dati binari codificati come stringa base64 in formato binario DynamoDB. Questo restituisce un oggetto che descrive il valore dell'attributo DynamoDB.

```
Input:      util.dynamodb.toBinary("foo")
Output:     { "B" : "foo" }
```

util.dynamodb.toBinarySet(List<String>)

Converte un elenco di dati binari codificati come stringhe base64 in formato set binario DynamoDB. Questo restituisce un oggetto che descrive il valore dell'attributo DynamoDB.

```
Input:      util.dynamodb.toBinarySet([ "foo", "bar", "baz" ])
Output:     { "BS" : [ "foo", "bar", "baz" ] }
```

ToBoolean utils

Elenco degli strumenti ToBoolean

util.dynamodb.toBoolean(Boolean)

Converte un booleano nel formato booleano DynamoDB appropriato. Questo restituisce un oggetto che descrive il valore dell'attributo DynamoDB.

```
Input:      util.dynamodb.toBoolean(true)
Output:     { "BOOL" : true }
```

ToNull utils

Elenco di utilità ToNull

util.dynamodb.toNull()

Restituisce un valore null nel formato null di DynamoDB. Questo restituisce un oggetto che descrive il valore dell'attributo DynamoDB.

```
Input:      util.dynamodb.toNull()
Output:     { "NULL" : null }
```

Utilità ToList

Elenco di utilità ToList

util.dynamodb.toList(List)

Converte un elenco di oggetti nel formato elenco DynamoDB. Ogni elemento dell'elenco viene inoltre convertito nel formato DynamoDB appropriato. Rappresenta alcuni oggetti nidificati in un determinato modo. Ad esempio, usa elenchi ("L") invece di set ("SS", "NS", "BS"). Questo restituisce un oggetto che descrive il valore dell'attributo DynamoDB.

```
Input:      util.dynamodb.toList([ "foo", 123, { "bar" : "baz" } ])
Output:     {
        "L" : [
            { "S" : "foo" },
```



```

        { "N" : 123 },
        {
            "M" : {
                "bar" : { "S" : "baz" }
            }
        }
    ]
}

```

Utilità ToMap

Elenco di utilità ToMap

util.dynamodb.toMap(Map)

Converte una mappa nel formato di mappa DynamoDB. Ogni valore nella mappa viene inoltre convertito nel formato DynamoDB appropriato. Rappresenta alcuni oggetti nidificati in un determinato modo. Ad esempio, usa elenchi ("L") invece di set ("SS", "NS", "BS"). Questo restituisce un oggetto che descrive il valore dell'attributo DynamoDB.

```

Input:      util.dynamodb.toMap({ "foo": "bar", "baz" : 1234, "beep": [ "boop"] })
Output:     {
              "M" : {
                  "foo"  : { "S" : "bar" },
                  "baz"  : { "N" : 1234 },
                  "beep" : {
                      "L" : [
                          { "S" : "boop" }
                      ]
                  }
              }
          }

```

util.dynamodb.toMapValues(Map)

Crea una copia della mappa in cui ogni valore è stato convertito nel formato DynamoDB appropriato. Rappresenta alcuni oggetti nidificati in un determinato modo. Ad esempio, usa elenchi ("L") invece di set ("SS", "NS", "BS").

```

Input:      util.dynamodb.toMapValues({ "foo": "bar", "baz" : 1234, "beep":
          [ "boop"] })

```

```
Output:  {
    "foo" : { "S" : "bar" },
    "baz" : { "N" : 1234 },
    "beep" : {
        "L" : [
            { "S" : "boop" }
        ]
    }
}
```

Note

Questo è leggermente diverso dal fatto che restituisce solo il contenuto del valore dell'attributo DynamoDB, ma non l'intero valore dell'attributo `util.dynamodb.toMap(Map)` stesso. Ad esempio, le istruzioni seguenti sono identiche:

```
util.dynamodb.toMapValues(<map>)
util.dynamodb.toMap(<map>)("M")
```

Utilità S3Object

Elenco delle utilità di S3Object

`util.dynamodb.toS3Object(String key, String bucket, String region)`

Converte la chiave, il bucket e la regione nella rappresentazione dell'oggetto DynamoDB S3. Questo restituisce un oggetto che descrive il valore dell'attributo DynamoDB.

```
Input:      util.dynamodb.toS3Object("foo", "bar", region = "baz")
Output:     { "S" : "{ \"s3\" : { \"key\" : \"foo\", \"bucket\" : \"bar\", \"region\" : \"baz\" } }" }
```

`util.dynamodb.toS3Object(String key, String bucket, String region, String version)`

Converte la chiave, il bucket, la regione e la versione opzionale nella rappresentazione dell'oggetto DynamoDB S3. Questo restituisce un oggetto che descrive il valore dell'attributo DynamoDB.

```
Input:      util.dynamodb.toS3Object("foo", "bar", "baz", "beep")
Output:     { "S" : "{ \"s3\" : { \"key\" : \"foo\", \"bucket\" : \"bar\", \"region\" : \"baz\", \"version\" = \"beep\" } }" }
```

util.dynamodb.fromS3ObjectJson(String)

Accetta il valore stringa di un oggetto DynamoDB S3 e restituisce una mappa che contiene la chiave, il bucket, la regione e la versione opzionale.

```
Input:      util.dynamodb.fromS3ObjectJson({ "S" : "{ \"s3\" : { \"key\" : \"foo\", \"bucket\" : \"bar\", \"region\" : \"baz\", \"version\" = \"beep\" } }" })
Output:     { "key" : "foo", "bucket" : "bar", "region" : "baz", "version" : "beep" }
```

Helper HTTP in util.http

L'`util.http` utilità fornisce metodi di supporto che è possibile utilizzare per gestire i parametri di richiesta HTTP e aggiungere intestazioni di risposta.

elenco di utilità util.http

util.http.copyHeaders(headers)

Copia le intestazioni dalla mappa, escluse le seguenti intestazioni HTTP con restrizioni:

- transfer-encoding
- connection
- host
- expect
- keep-alive
- upgrade
- proxy-authenticate
- proxy-authorization
- te
- content-length

util.http.addResponseHeader(String, Object)

Aggiunge una singola intestazione personalizzata con il nome (`String`) e il valore (`Object`) della risposta. Si applicano le limitazioni seguenti:

- Oltre all'elenco delle intestazioni con restrizioni per `copyHeaders(headers)`, i nomi delle intestazioni non possono corrispondere a nessuno dei seguenti:
 - Access-Control-Allow-Credentials
 - Access-Control-Allow-Origin
 - Access-Control-Expose-Headers
 - Access-Control-Max-Age
 - Access-Control-Allow-Methods
 - Access-Control-Allow-Headers
 - Vary
 - Content-Type
- I nomi delle intestazioni non possono iniziare con i prefissi limitati o. `x-amzn-` o `x-amz-`
- La dimensione delle intestazioni di risposta personalizzate non può superare i 4 KB. Sono inclusi i nomi e i valori delle intestazioni.
- È necessario definire ogni intestazione di risposta una volta per operazione GraphQL. Tuttavia, se definisci più volte un'intestazione personalizzata con lo stesso nome, nella risposta viene visualizzata la definizione più recente. Tutte le intestazioni vengono conteggiate ai fini del limite di dimensione dell'intestazione indipendentemente dalla denominazione.
- Le intestazioni con un nome vuoto o limitato (`String`) o un valore nullo verranno ignorate e (`Object`) genereranno un `ResponseHeaderError` errore che viene aggiunto all'output dell'operazione. `errors`

```
export function request(ctx) {
  util.http.addResponseHeader('itemsCount', 7)
  util.http.addResponseHeader('render', ctx.args.render)
  return {}
}
```

util.http.addResponseHeaders(Map)

Aggiunge più intestazioni di risposta alla risposta dalla mappa di nomi e valori specificata. (`String`) (`Object`) Le stesse limitazioni elencate per il `addResponseHeader(String, Object)` metodo si applicano anche a questo metodo.

```
export function request(ctx) {
  const headers = {
    headerInt: 12,
    headerString: 'stringValue',
    headerObject: {
      field1: 7,
      field2: 'string'
    }
  }
  util.http.addResponseHeaders(headers)
  return {}
}
```

Aiutanti di trasformazione in util.transform

`util.transform` contiene metodi di supporto che semplificano l'esecuzione di operazioni complesse su fonti di dati.

Elenco di utilità degli aiutanti di trasformazione

`util.transform.toDynamoDBFilterExpression(filterObject: DynamoDBFilterObject) : string`

Converte una stringa di input in un'espressione di filtro da utilizzare con DynamoDB. Si consiglia l'utilizzo `toDynamoDBFilterExpression` con le [funzioni integrate](#) del modulo.

`util.transform.toElasticsearchQueryDSL(object: OpenSearchQueryObject) : string`

Converte l'input dato nella sua espressione OpenSearch Query DSL equivalente, restituendola come stringa JSON.

Esempio di input:

```
util.transform.toElasticsearchQueryDSL({
  "upvotes":{
    "ne":15,
    "range":[
      10,
      20
    ]
  }
})
```

```

    },
    "title":{
      "eq":"hihihi",
      "wildcard":"h*i"
    }
  })

```

Output di esempio:

```

{
  "bool":{
    "must":[
      {
        "bool":{
          "must":[
            {
              "bool":{
                "must_not":{
                  "term":{
                    "upvotes":15
                  }
                }
              }
            }
          ],
          },
      {
        "range":{
          "upvotes":{
            "gte":10,
            "lte":20
          }
        }
      }
    ]
  },
  {
    "bool":{
      "must":[
        {
          "term":{
            "title":"hihihi"
          }
        }
      ],
    },
  }
}

```

```
{
  "wildcard": {
    "title": "h*i"
  }
}
```

Note

Si presume che l'operatore predefinito sia AND.

`util.transform.toSubscriptionFilter(objFilter, ignoredFields?, rules?): SubscriptionFilter`

Converte un oggetto Map di input in un oggetto SubscriptionFilter espressione. Il `util.transform.toSubscriptionFilter` metodo viene utilizzato come input per l'`extensions.setSubscriptionFilter()` estensione. Per ulteriori informazioni, consulta [Estensioni](#).

Note

I parametri e l'istruzione return sono elencati di seguito:

Parametri

- `objFilter: SubscriptionFilterObject`

Un oggetto Map di input che viene convertito nell'oggetto SubscriptionFilter espressione.

- `ignoredFields: SubscriptionFilterExcludeKeysType` (opzionale)

Uno List dei nomi di campo nel primo oggetto che verrà ignorato.

- `rules: SubscriptionFilterRuleObject` (opzionale)

Un oggetto Map di input con regole rigorose che viene incluso durante la costruzione dell'oggetto SubscriptionFilter espressione. Queste regole rigorose verranno

include nell'oggetto `SubscriptionFilter` espressione in modo che almeno una delle regole venga soddisfatta per passare il filtro di sottoscrizione.

Risposta

Restituisce una [SubscriptionFilter](#).

util.transform.toSubscriptionFilter(Map, List)

Converte un oggetto Map di input in un oggetto `SubscriptionFilter` espressione. Il `util.transform.toSubscriptionFilter` metodo viene utilizzato come input per l'`extensions.setSubscriptionFilter()` estensione. Per ulteriori informazioni, consulta [Estensioni](#).

Il primo argomento è l'oggetto Map di input che viene convertito nell'oggetto `SubscriptionFilter` espressione. Il secondo argomento riguarda i nomi List di campo che vengono ignorati nel primo oggetto Map di input durante la costruzione dell'oggetto `SubscriptionFilter` espressione.

util.transform.toSubscriptionFilter(Map, List, Map)

Converte un oggetto Map di input in un `SubscriptionFilter` oggetto espressione. Il `util.transform.toSubscriptionFilter` metodo viene utilizzato come input per l'`extensions.setSubscriptionFilter()` estensione. Per ulteriori informazioni, consulta [Estensioni](#).

util.transform.toDynamoDBConditionExpression(conditionObject)

Crea un'espressione di condizione DynamoDB.

Argomenti del filtro di abbonamento

La tabella seguente spiega come vengono definiti gli argomenti delle seguenti utilità:

- `Util.transform.toSubscriptionFilter(objFilter, ignoredFields?, rules?): SubscriptionFilter`

Argument 1: Map

L'argomento 1 è un Map oggetto con i seguenti valori chiave:

- nomi di campo

- «e»
- «o»

Per i nomi di campo come chiavi, le condizioni sulle voci di questi campi sono nel formato di.

```
"operator" : "value"
```

L'esempio seguente mostra come aggiungere voci a: Map

```
"field_name" : {
    "operator1" : value
}

## We can have multiple conditions for the same field_name:

"field_name" : {
    "operator1" : value
    "operator2" : value
    .
    .
    .
}
```

Quando un campo contiene due o più condizioni, si considera che tutte queste condizioni utilizzino l'operazione OR.

L'input Map può anche avere «e» e «or» come chiavi, il che implica che tutte le voci al suo interno devono essere unite utilizzando la logica AND o OR a seconda della chiave. I valori chiave «and» e «or» prevedono una serie di condizioni.

```
"and" : [
    {
        "field_name1" : {
            "operator1" : value
        }
    },
    {
        "field_name2" : {
            "operator1" : value
        }
    }
]
```

```
    },
    .
    .
  ].
```

Nota che puoi annidare «and» e «or». Cioè, puoi aver annidato «e» /"or» all'interno di un altro blocco «e» /"or». Tuttavia, questo non funziona per campi semplici.

```
"and" : [
  {
    "field_name1" : {
      "operator" : value
    }
  },
  {
    "or" : [
      {
        "field_name2" : {
          "operator" : value
        }
      },
      {
        "field_name3" : {
          "operator" : value
        }
      }
    ]
  }
].
```

L'esempio seguente mostra un input dell'argomento 1
 utilizzando `util.transform.toSubscriptionFilter(Map) : Map`.

Ingresso/i

Argomento 1: Mappa:

```
{
  "percentageUp": {
    "lte": 50,
    "gte": 20
```

```
},
"and": [
  {
    "title": {
      "ne": "Book1"
    }
  },
  {
    "downvotes": {
      "gt": 2000
    }
  }
],
"or": [
  {
    "author": {
      "eq": "Admin"
    }
  },
  {
    "isPublished": {
      "eq": false
    }
  }
]
}
```

Output

Il risultato è un Map oggetto:

```
{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "percentageUp",
          "operator": "lte",
          "value": 50
        },
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        }
      ]
    }
  ]
}
```

```
    },
    {
      "fieldName": "downvotes",
      "operator": "gt",
      "value": 2000
    },
    {
      "fieldName": "author",
      "operator": "eq",
      "value": "Admin"
    }
  ]
},
{
  "filters": [
    {
      "fieldName": "percentageUp",
      "operator": "lte",
      "value": 50
    },
    {
      "fieldName": "title",
      "operator": "ne",
      "value": "Book1"
    },
    {
      "fieldName": "downvotes",
      "operator": "gt",
      "value": 2000
    },
    {
      "fieldName": "isPublished",
      "operator": "eq",
      "value": false
    }
  ]
},
{
  "filters": [
    {
      "fieldName": "percentageUp",
      "operator": "gte",
      "value": 20
    },
  ],
```

```
{
  {
    "fieldName": "title",
    "operator": "ne",
    "value": "Book1"
  },
  {
    "fieldName": "downvotes",
    "operator": "gt",
    "value": 2000
  },
  {
    "fieldName": "author",
    "operator": "eq",
    "value": "Admin"
  }
]
},
{
  "filters": [
    {
      "fieldName": "percentageUp",
      "operator": "gte",
      "value": 20
    },
    {
      "fieldName": "title",
      "operator": "ne",
      "value": "Book1"
    },
    {
      "fieldName": "downvotes",
      "operator": "gt",
      "value": 2000
    },
    {
      "fieldName": "isPublished",
      "operator": "eq",
      "value": false
    }
  ]
}
]
```

Argument 2: List

L'argomento 2 contiene una `List` serie di nomi di campo che non devono essere considerati nell'input Map (argomento 1) durante la costruzione dell'oggetto `SubscriptionFilter` espressione. `List` Possono anche essere vuoti.

L'esempio seguente mostra gli input dell'argomento 1 e dell'argomento 2 utilizzando `util.transform.toSubscriptionFilter(Map, List) : Map`.

Ingresso/i

Argomento 1: Mappa:

```
{
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "gt": 20
      }
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

Argomento 2: Elenco:

```
["percentageUp", "author"]
```

Output

Il risultato è un Map oggetto:

```
{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 20
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
        }
      ]
    }
  ]
}
```

Argument 3: Map

L'argomento 3 è un Map oggetto che ha nomi di campo come valori chiave (non può avere «and» o «or»). Per i nomi di campo come chiavi, le condizioni in questi campi sono voci nel formato di "operator" : "value". A differenza dell'argomento 1, l'argomento 3 non può avere più condizioni nella stessa chiave. Inoltre, l'argomento 3 non contiene una clausola «and» o «or», quindi non è prevista nemmeno la nidificazione.

L'argomento 3 rappresenta un elenco di regole rigorose, che vengono aggiunte all'oggetto `SubscriptionFilter` espressione in modo che venga soddisfatta almeno una di queste condizioni per passare il filtro.

```
{
  "fieldname1": {
    "operator": value
  },
  "fieldname2": {
    "operator": value
  }
}
.
.
.
```

L'esempio seguente mostra gli input dell'argomento 1, dell'argomento 2 e dell'argomento 3 utilizzando `util.transform.toSubscriptionFilter(Map, List, Map) : Map`.

Ingresso/i

Argomento 1: Mappa:

```
{
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "lt": 20
      }
    }
  ],
  "or": [
    {
      "author": {
```



```
      "eq": "Admin"
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

Argomento 2: Elenco:

```
["percentageUp", "author"]
```

Argomento 3: Mappa:

```
{
  "upvotes": {
    "gte": 250
  },
  "author": {
    "eq": "Person1"
  }
}
```

Output

Il risultato è un Map oggetto:

```
{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 20
        }
      ]
    }
  ]
}
```

```
    },
    {
      "fieldName": "isPublished",
      "operator": "eq",
      "value": false
    },
    {
      "fieldName": "upvotes",
      "operator": "gte",
      "value": 250
    }
  ]
},
{
  "filters": [
    {
      "fieldName": "title",
      "operator": "ne",
      "value": "Book1"
    },
    {
      "fieldName": "downvotes",
      "operator": "gt",
      "value": 20
    },
    {
      "fieldName": "isPublished",
      "operator": "eq",
      "value": false
    },
    {
      "fieldName": "author",
      "operator": "eq",
      "value": "Person1"
    }
  ]
}
]
```

String helper in util.str

`util.str` contiene metodi per facilitare le operazioni più comuni sulle stringhe.

elenco di utilità `util.str`

`util.str.normalize(String, String)`

Normalizza una stringa utilizzando uno dei quattro moduli di normalizzazione Unicode: NFC, NFD, NFKC o NFKD. Il primo argomento è la stringa da normalizzare. Il secondo argomento è «nfc», «nfd», «nfkc» o «nfkd» e specifica il tipo di normalizzazione da utilizzare per il processo di normalizzazione.

Estensioni

`extensions` contiene una serie di metodi per eseguire azioni aggiuntive all'interno dei resolver.

Estensioni di memorizzazione nella cache

`extensions.evictFromApiCache(typeName: string, fieldName: string, keyValuePair: Record<string, any>) : Object`

Rimuove un elemento dalla cache lato server. AWS AppSync Il primo argomento è il nome del tipo. Il secondo argomento è il nome del campo. Il terzo argomento è un oggetto contenente elementi della coppia chiave-valore che specificano il valore della chiave di memorizzazione nella cache. È necessario inserire gli elementi nell'oggetto nello stesso ordine delle chiavi di memorizzazione nella cache del resolver memorizzato nella cache. `cachingKey` [Per ulteriori informazioni sulla memorizzazione nella cache, vedete Comportamento della memorizzazione nella cache.](#)

Esempio 1:

Questo esempio rimuove gli elementi che sono stati memorizzati nella cache di un resolver chiamato `Query.allClasses` su cui è stata utilizzata una chiave di memorizzazione nella cache chiamata `context.arguments.semester`. Quando viene chiamata la mutazione e il resolver viene eseguito, se una voce viene cancellata con successo, la risposta contiene un `apiCacheEntriesDeleted` valore nell'oggetto `extensions` che mostra quante voci sono state eliminate.

```
import { util, extensions } from '@aws-appsync/utils';
```

```
export const request = (ctx) => ({ payload: null });

export function response(ctx) {
  extensions.evictFromApiCache('Query', 'allClasses', {
    'context.arguments.semester': ctx.args.semester,
  });
  return null;
}
```

Note

Questa funzione funziona solo per le mutazioni, non per le interrogazioni.

Estensioni di abbonamento

`extensions.setSubscriptionFilter(filterJsonObject)`

Definisce filtri di abbonamento avanzati. Ogni evento di notifica di sottoscrizione viene valutato sulla base dei filtri di sottoscrizione forniti e invia notifiche ai clienti se tutti i filtri lo confermano. `true` L'argomento è `filterJsonObject` (ulteriori informazioni su questo argomento sono disponibili di seguito nella `filterJsonObject` sezione Argomento:). Vedi [Filtraggio avanzato degli abbonamenti](#).

Note

È possibile utilizzare questa funzione di estensione solo nel gestore delle risposte di un resolver di sottoscrizioni. Inoltre, ti consigliamo di utilizzarla `util.transform.toSubscriptionFilter` per creare il tuo filtro.

`extensions.setSubscriptionInvalidationFilter(filterJsonObject)`

Definisce i filtri di invalidazione dell'abbonamento. I filtri di sottoscrizione vengono valutati in base al payload di invalidazione, quindi invalidano un determinato abbonamento se i filtri restituiscono lo stesso risultato. `true` L'argomento è `filterJsonObject` (ulteriori informazioni su questo argomento sono disponibili più avanti nella sezione Argomento:). `filterJsonObject` Vedi [Filtraggio avanzato degli abbonamenti](#).

Note

È possibile utilizzare questa funzione di estensione solo nel gestore delle risposte di un resolver di sottoscrizioni. Inoltre, ti consigliamo di utilizzarla `util.transform.toSubscriptionFilter` per creare il tuo filtro.

`extensions.invalidateSubscriptions(invalidationJsonObject)`

Utilizzato per avviare l'invalidazione dell'abbonamento a seguito di una mutazione. L'argomento è `invalidationJsonObject` (ulteriori informazioni su questo argomento sono disponibili di seguito nella sezione Argomento: `invalidationJsonObject`).

Note

Questa estensione può essere utilizzata solo nei modelli di mappatura delle risposte dei risolutori di mutazioni.

È possibile utilizzare al massimo cinque chiamate di `extensions.invalidateSubscriptions()` metodo uniche in ogni singola richiesta. Se superi questo limite, riceverai un errore GraphQL.

Argomento: `filterJsonObject`

L'oggetto JSON definisce i filtri di sottoscrizione o di invalidazione. È una serie di filtri in un `filterGroup`. Ogni filtro è una raccolta di filtri individuali.

```
{
  "filterGroup": [
    {
      "filters" : [
        {
          "fieldName" : "userId",
          "operator" : "eq",
          "value" : 1
        }
      ]
    }
  ],
  {
```

```

      "filters" : [
        {
          "fieldName" : "group",
          "operator" : "in",
          "value" : ["Admin", "Developer"]
        }
      ]
    }
  ]
}

```

Ogni filtro ha tre attributi:

- **fieldName**— Il campo dello schema GraphQL.
- **operator**— Il tipo di operatore.
- **value**— I valori da confrontare con il **fieldName** valore di notifica dell'abbonamento.

Di seguito è riportato un esempio di assegnazione di questi attributi:

```

{
  "fieldName" : "severity",
  "operator" : "le",
  "value" : context.result.severity
}

```

Argomento: invalidationJsonObject

invalidationJsonObject Definisce quanto segue:

- **subscriptionField**— L'abbonamento allo schema GraphQL da invalidare. Un singolo abbonamento, definito come una stringa **subscriptionField**, viene considerato invalidato.
- **payload**— Un elenco di coppie chiave-valore che viene utilizzato come input per invalidare le sottoscrizioni se il filtro di invalidazione valuta in base ai relativi valori. **true**

L'esempio seguente invalida i client sottoscritti e connessi che utilizzano l'abbonamento quando il filtro di invalidazione definito nel resolver di **onUserDelete** sottoscrizione restituisce un risultato conforme al valore. **true payload**

```
export const request = (ctx) => ({ payload: null });
```

```
export function response(ctx) {
  extensions.invalidateSubscriptions({
    subscriptionField: 'onUserDelete',
    payload: { group: 'Developer', type: 'Full-Time' },
  });
  return ctx.result;
}
```

Helper XML in util.xml

`util.xml` contiene metodi per facilitare la conversione di stringhe XML.

elenco di utilità di `util.xml`

`util.xml.toMap(String)` : Object

Converte una stringa XML in un dizionario.

Esempio 1:

Input:

```
<?xml version="1.0" encoding="UTF-8"?>
<posts>
<post>
  <id>1</id>
  <title>Getting started with GraphQL</title>
</post>
</posts>
```

Output (object):

```
{
  "posts": {
    "post": {
      "id": 1,
      "title": "Getting started with GraphQL"
    }
  }
}
```

Esempio 2:

Input:

```
<?xml version="1.0" encoding="UTF-8"?>
<posts>
<post>
  <id>1</id>
  <title>Getting started with GraphQL</title>
</post>
<post>
  <id>2</id>
  <title>Getting started with AppSync</title>
</post>
</posts>
```

Output (JavaScript object):

```
{
  "posts":{
    "post":[
      {
        "id":1,
        "title":"Getting started with GraphQL"
      },
      {
        "id":2,
        "title":"Getting started with AppSync"
      }
    ]
  }
}
```

util.xml.toJsonString(String, Boolean?) : String

Converte una stringa XML in una stringa JSON. È simile a `toMap`, tranne per il fatto che l'output è una stringa. Questa funzione è utile se si desidera convertire direttamente e restituire la risposta XML da un oggetto HTTP in formato JSON. È possibile impostare un parametro booleano opzionale per determinare se si desidera codificare il JSON come stringa.

AWS AppSync JavaScript riferimento alla funzione resolver per DynamoDB

La funzione AWS AppSync DynamoDB ti consente di utilizzare [GraphQL](#) per archiviare e recuperare dati nelle tabelle Amazon DynamoDB esistenti nel tuo account mappando una richiesta GraphQL in entrata in una chiamata DynamoDB e quindi mappando la risposta DynamoDB a GraphQL. Questa sezione descrive i gestori di richieste e risposte per le operazioni DynamoDB supportate:

- [GetItem](#) - La GetItem richiesta consente di indicare alla funzione DynamoDB di effettuare una GetItem richiesta a DynamoDB e di specificare la chiave dell'elemento in DynamoDB e se utilizzare o meno una lettura coerente.
- [PutItem](#) - Il documento di mappatura della PutItem richiesta consente di indicare alla funzione DynamoDB di PutItem effettuare una richiesta a DynamoDB e di specificare la chiave dell'elemento in DynamoDB, il contenuto completo dell'elemento (composto da key e attributeValues) e le condizioni per il successo dell'operazione.
- [UpdateItem](#) - La UpdateItem richiesta consente di indicare alla funzione DynamoDB di effettuare una UpdateItem richiesta a DynamoDB e consente di specificare la chiave dell'elemento in DynamoDB, un'espressione di aggiornamento che descrive come aggiornare l'elemento in DynamoDB e le condizioni per il successo dell'operazione.
- [DeleteItem](#) - La DeleteItem richiesta consente di indicare alla funzione DynamoDB di effettuare una DeleteItem richiesta a DynamoDB e di specificare la chiave dell'elemento in DynamoDB e le condizioni per il successo dell'operazione.
- [Query](#): l'oggetto Query request consente di indicare al resolver DynamoDB di effettuare una richiesta Query a DynamoDB e di specificare l'espressione chiave, l'indice da utilizzare, i filtri aggiuntivi, il numero di elementi da restituire, se utilizzare letture coerenti, la direzione della query (avanti o indietro) e i token di impaginazione.
- [Scansione](#): la richiesta Scan consente di indicare alla funzione DynamoDB di effettuare una richiesta di scansione a DynamoDB e di specificare un filtro per escludere i risultati, quale indice utilizzare, quanti elementi restituire, se utilizzare letture coerenti, token di impaginazione e scansioni parallele.
- [Sincronizzazione](#): l'oggetto di richiesta di sincronizzazione consente di recuperare tutti i risultati da una tabella DynamoDB e quindi ricevere solo i dati modificati dall'ultima query (gli aggiornamenti delta). Le richieste di sincronizzazione possono essere effettuate solo su sorgenti dati DynamoDB con versione. È possibile specificare un filtro per escludere i risultati, il numero di elementi da restituire, i token di paginazione e quando è stata avviata l'ultima operazione di sincronizzazione.

- [BatchGetItem](#) - L'oggetto BatchGetItem request consente di indicare alla funzione DynamoDB di effettuare una BatchGetItem richiesta a DynamoDB per recuperare più elementi, potenzialmente su più tabelle. Per questo oggetto di richiesta, è necessario specificare i nomi delle tabelle da cui recuperare gli elementi e le chiavi degli elementi da recuperare da ciascuna tabella.
- [BatchDeleteItem](#) - L'oggetto BatchDeleteItem request consente di indicare alla funzione DynamoDB di fare una BatchWriteItem richiesta a DynamoDB per eliminare più elementi, potenzialmente su più tabelle. Per questo oggetto di richiesta, è necessario specificare i nomi delle tabelle da cui eliminare gli elementi e le chiavi degli elementi da eliminare da ciascuna tabella.
- [BatchPutItem](#) - L'oggetto BatchPutItem request consente di indicare alla funzione DynamoDB di fare una BatchWriteItem richiesta a DynamoDB per inserire più elementi, potenzialmente su più tabelle. Per questo oggetto di richiesta, è necessario specificare i nomi delle tabelle in cui inserire gli elementi e gli elementi completi da inserire in ogni tabella.
- [TransactGetItems](#) - L'oggetto TransactGetItems request consente di indicare alla funzione DynamoDB di effettuare una TransactGetItems richiesta a DynamoDB per recuperare più elementi, potenzialmente su più tabelle. Per questo oggetto di richiesta, è necessario specificare il nome della tabella di ogni elemento della richiesta da cui recuperare l'elemento e la chiave di ogni elemento della richiesta da recuperare da ogni tabella.
- [TransactWriteItems](#) - L'oggetto TransactWriteItems request consente di indicare alla funzione DynamoDB di TransactWriteItems richiedere a DynamoDB di scrivere più elementi, potenzialmente su più tabelle. Per questo oggetto di richiesta, è necessario specificare il nome della tabella di destinazione di ogni elemento della richiesta, l'operazione di ogni elemento della richiesta da eseguire e la chiave di ogni elemento della richiesta da scrivere.
- [Sistema dei tipi \(mappatura delle richieste\)](#): scopri di più su come la digitazione DynamoDB è integrata nelle richieste. AWS AppSync
- [Sistema di tipi \(mappatura delle risposte\)](#): scopri di più su come i tipi DynamoDB vengono convertiti automaticamente in GraphQL o JSON in un payload di risposta.
- [Filtri](#): scopri di più sui filtri per le operazioni di interrogazione e scansione.
- [Espressioni](#) condizionali: scopri di più sulle espressioni di PutItem condizione e DeleteItem sulle operazioni. UpdateItem
- [Espressioni delle condizioni delle transazioni](#): ulteriori informazioni sulle espressioni di condizione per TransactWriteItems le operazioni.
- [Proiezioni](#): ulteriori informazioni su come specificare gli attributi nelle operazioni di lettura.

GetItem

La `GetItem` richiesta consente di indicare alla funzione AWS AppSync DynamoDB di effettuare una `GetItem` richiesta a DynamoDB e consente di specificare:

- La chiave dell'elemento in DynamoDB
- Se utilizzare una lettura consistente o no

La `GetItem` richiesta ha la seguente struttura:

```
type DynamoDBGetItem = {  
  operation: 'GetItem';  
  key: { [key: string]: any };  
  consistentRead?: ConsistentRead;  
  projection?: {  
    expression: string;  
    expressionNames?: { [key: string]: string };  
  };  
};
```

I campi sono definiti come segue:

GetItem campi

GetItem elenco dei campi

operation

L'operazione DynamoDB da eseguire. Per eseguire l'operazione `GetItem` DynamoDB, il valore deve essere impostato su `GetItem`. Questo valore è obbligatorio.

key

La chiave dell'elemento in DynamoDB. Gli elementi DynamoDB possono avere una sola chiave hash o una chiave hash e una chiave di ordinamento, a seconda della struttura della tabella. Per ulteriori informazioni su come specificare un «valore digitato», consulta [Sistema di tipi \(mappatura delle richieste\)](#). Questo valore è obbligatorio.

consistentRead

Se eseguire o meno una lettura fortemente coerente con DynamoDB. Si tratta di un'opzione facoltativa, impostata di default su `false`.

projection

Una proiezione utilizzata per specificare gli attributi da restituire dall'operazione DynamoDB. [Per ulteriori informazioni sulle proiezioni, vedere Proiezioni](#). Questo campo è facoltativo.

L'elemento restituito da DynamoDB viene automaticamente convertito in tipi primitivi GraphQL e JSON ed è disponibile nel contesto `result()`. `context.result`

Per ulteriori informazioni sulla conversione dei tipi in DynamoDB, [vedere Sistema dei tipi](#) (mappatura delle risposte).

[Per ulteriori informazioni sui JavaScript resolver, consulta la panoramica dei resolver. JavaScript](#)

Esempio

L'esempio seguente è un gestore di richieste di funzioni per una query `getThing(foo: String!, bar: String!)` GraphQL:

```
export function request(ctx) {
  const {foo, bar} = ctx.args
  return {
    operation : "GetItem",
    key : util.dynamodb.toMapValues({foo, bar}),
    consistentRead : true
  }
}
```

Per ulteriori informazioni sull'API `GetItem` di DynamoDB, consulta la [documentazione API di DynamoDB](#).

PutItem

Il documento di mappatura delle `PutItem` richieste consente di indicare alla AWS AppSync funzione DynamoDB di `PutItem` effettuare una richiesta a DynamoDB e consente di specificare quanto segue:

- La chiave dell'elemento in DynamoDB
- L'intero contenuto della voce (costituita da `key` e `attributeValues`)
- Condizioni per la riuscita dell'operazione

La PutItem richiesta ha la seguente struttura:

```
type DynamoDBPutItemRequest = {  
  operation: 'PutItem';  
  key: { [key: string]: any };  
  attributeValues: { [key: string]: any };  
  condition?: ConditionCheckExpression;  
  customPartitionKey?: string;  
  populateIndexFields?: boolean;  
  _version?: number;  
};
```

I campi sono definiti come segue:

PutItem campi

PutItem elenco dei campi

operation

L'operazione DynamoDB da eseguire. Per eseguire l'operazione PutItem DynamoDB, il valore deve essere impostato su PutItem. Questo valore è obbligatorio.

key

La chiave dell'elemento in DynamoDB. Gli elementi DynamoDB possono avere una sola chiave hash o una chiave hash e una chiave di ordinamento, a seconda della struttura della tabella. Per ulteriori informazioni su come specificare un «valore digitato», consulta [Sistema di tipi \(mappatura delle richieste\)](#). Questo valore è obbligatorio.

attributeValues

Gli altri attributi della voce da inserir in DynamoDB. Per ulteriori informazioni su come specificare un «valore digitato», vedete [Type system \(request mapping\)](#). Questo campo è facoltativo.

condition

Una condizione per determinare se la richiesta deve riuscire o no in base allo stato dell'oggetto già incluso in DynamoDB. Se non viene specificata alcuna condizione, la richiesta PutItem sovrascrive qualsiasi valore esistente per quella voce. Per ulteriori informazioni sulle condizioni, vedere Espressioni di [condizione](#). Questo valore è facoltativo.

`_version`

Valore numerico che rappresenta l'ultima versione nota di un elemento. Questo valore è facoltativo. Questo campo viene utilizzato per il rilevamento dei conflitti ed è supportato solo nelle origini dati con versione.

`customPartitionKey`

Se abilitato, questo valore di stringa modifica il formato dei ds_pk record ds_sk and utilizzati dalla tabella delta sync quando il controllo delle versioni è abilitato (per ulteriori informazioni, consulta [Conflict detection and sync](#) nella AWS AppSync Developer Guide). Se abilitata, è abilitata anche l'elaborazione della populateIndexFields voce. Questo campo è facoltativo.

`populateIndexFields`

Un valore booleano che, se abilitato insieme a customPartitionKey, crea nuove voci per ogni record nella tabella delta sync, in particolare nelle colonne gsi_ds_pk and gsi_ds_sk. Per ulteriori informazioni, consulta [Rilevamento e sincronizzazione dei conflitti](#) nella Guida per gli AWS AppSync sviluppatori. Questo campo è facoltativo.

L'elemento scritto in DynamoDB viene automaticamente convertito in tipi primitivi GraphQL e JSON ed è disponibile nel contesto `result()`. `context.result`

L'elemento scritto in DynamoDB viene automaticamente convertito in tipi primitivi GraphQL e JSON ed è disponibile nel contesto `result()`. `context.result`

Per ulteriori informazioni sulla conversione dei tipi in DynamoDB, [vedere Sistema dei tipi](#) (mappatura delle risposte).

[Per ulteriori informazioni sui JavaScript resolver, consulta la panoramica dei resolver. JavaScript](#)

Esempio 1

L'esempio seguente è un gestore di richieste di funzioni per una mutazione `updateThing(foo: String!, bar: String!, name: String!, version: Int!)` GraphQL.

Se non esiste alcuna voce con la chiave specificata, viene creata. Se esiste, viene sovrascritta.

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
```

```
const { foo, bar, ...values } = ctx.args
return {
  operation: 'PutItem',
  key: util.dynamodb.toMapValues({foo, bar}),
  attributeValues: util.dynamodb.toMapValues(values),
};
}
```

Esempio 2

L'esempio seguente è un gestore di richieste di funzioni per una mutazione `updateThing(foo: String!, bar: String!, name: String!, expectedVersion: Int!)` GraphQL.

Questo esempio verifica che l'elemento attualmente in DynamoDB abbia `version` il campo impostato su `expectedVersion`

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { foo, bar, name, expectedVersion } = ctx.args;
  const values = { name, version: expectedVersion + 1 };
  let condition = util.transform.toDynamoDBConditionExpression({
    version: { eq: expectedVersion },
  });

  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({ foo, bar }),
    attributeValues: util.dynamodb.toMapValues(values),
    condition,
  };
}
```

Per ulteriori informazioni sull'API `PutItem` di DynamoDB, consulta la [documentazione API di DynamoDB](#).

UpdateItem

La `UpdateItem` richiesta consente di indicare alla funzione AWS AppSync DynamoDB di effettuare una `UpdateItem` richiesta a DynamoDB e consente di specificare quanto segue:

- La chiave dell'elemento in DynamoDB

- Un'espressione di aggiornamento che descrive come aggiornare l'elemento in DynamoDB
- Condizioni per la riuscita dell'operazione

La `UpdateItem` richiesta ha la seguente struttura:

```
type DynamoDBUpdateItemRequest = {  
  operation: 'UpdateItem';  
  key: { [key: string]: any };  
  update: {  
    expression: string;  
    expressionNames?: { [key: string]: string };  
    expressionValues?: { [key: string]: any };  
  };  
  condition?: ConditionCheckExpression;  
  customPartitionKey?: string;  
  populateIndexFields?: boolean;  
  _version?: number;  
};
```

I campi sono definiti come segue:

UpdateItem campi

UpdateItem elenco dei campi

operation

L'operazione DynamoDB da eseguire. Per eseguire l'operazione `UpdateItem` DynamoDB, il valore deve essere impostato su `UpdateItem`. Questo valore è obbligatorio.

key

La chiave dell'elemento in DynamoDB. Gli elementi DynamoDB possono avere una sola chiave hash o una chiave hash e una chiave di ordinamento, a seconda della struttura della tabella.

[Per ulteriori informazioni sulla specificazione di un «valore digitato», consulta Sistema di tipi \(mappatura delle richieste\)](#). Questo valore è obbligatorio.

update

La `update` sezione consente di specificare un'espressione di aggiornamento che descrive come aggiornare l'elemento in DynamoDB. Per ulteriori informazioni su come scrivere espressioni di

aggiornamento, consulta la documentazione di [UpdateExpressions DynamoDB](#). Questa sezione è obbligatoria.

La sezione `update` ha tre componenti:

`expression`

L'espressione di aggiornamento. Questo valore è obbligatorio.

`expressionNames`

Le sostituzioni per i segnaposto dell'attributo di espressione name sotto forma di coppie chiave-valore. La chiave corrisponde a un segnaposto per il nome utilizzato in `expression`, e il valore deve essere una stringa corrispondente al nome dell'attributo dell'elemento in DynamoDB. Questo è un campo facoltativo in cui vanno riportate solo le sostituzioni per i segnaposto dell'attributo di espressione name utilizzate in `expression`.

`expressionValues`

Le sostituzioni per i segnaposto dell'attributo di espressione value sotto forma di coppie chiave-valore. La chiave corrisponde a un segnaposto per un valore utilizzato in `expression`, mentre il valore deve essere un valore tipizzato. Per ulteriori informazioni su come specificare un «valore digitato», consulta [Type system](#) (request mapping). Questo elemento deve essere specificato. Questo è un campo facoltativo in cui vanno riportate solo le sostituzioni per i segnaposto dell'attributo di espressione value utilizzate in `expression`.

`condition`

Una condizione per determinare se la richiesta deve riuscire o no in base allo stato dell'oggetto già incluso in DynamoDB. Se non viene specificata alcuna condizione, la richiesta `UpdateItem` aggiorna qualsiasi valore esistente, indipendentemente dal suo stato attuale. Per ulteriori informazioni sulle condizioni, vedere Espressioni di [condizione](#). Questo valore è facoltativo.

`_version`

Valore numerico che rappresenta l'ultima versione nota di un elemento. Questo valore è facoltativo. Questo campo viene utilizzato per il rilevamento dei conflitti ed è supportato solo nelle origini dati con versione.

`customPartitionKey`

Se abilitato, questo valore di stringa modifica il formato dei `ds_pk` record `ds_sk` and utilizzati dalla tabella delta sync quando il controllo delle versioni è abilitato (per ulteriori informazioni,

consulta [Conflict detection and sync](#) nella AWS AppSync Developer Guide). Se abilitata, è abilitata anche l'elaborazione della `populateIndexFields` voce. Questo campo è facoltativo.

populateIndexFields

Un valore booleano che, se abilitato insieme a `customPartitionKey`, crea nuove voci per ogni record nella tabella delta sync, in particolare nelle colonne `gsi_ds_pk` and `gsi_ds_sk`. Per ulteriori informazioni, consulta [Rilevamento e sincronizzazione dei conflitti](#) nella Guida per gli AWS AppSync sviluppatori. Questo campo è facoltativo.

L'elemento aggiornato in DynamoDB viene automaticamente convertito in tipi primitivi GraphQL e JSON ed è disponibile nel contesto `result()`. `context.result`

Per ulteriori informazioni sulla conversione dei tipi in DynamoDB, [vedere Sistema dei tipi](#) (mappatura delle risposte).

[Per ulteriori informazioni sui JavaScript resolver, consulta la panoramica dei resolver. JavaScript](#)

Esempio 1

L'esempio seguente è un gestore di richieste di funzioni per la mutazione `upvote(id: ID!)` GraphQL.

In questo esempio, un elemento in DynamoDB ha i `upvotes` suoi campi `version` e incrementati di 1.

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { id } = ctx.args;
  return {
    operation: 'UpdateItem',
    key: util.dynamodb.toMapValues({ id }),
    update: {
      expression: 'ADD #voteField :plusOne, version :plusOne',
      expressionNames: { '#voteField': 'upvotes' },
      expressionValues: { ':plusOne': { N: 1 } },
    },
  },
};
}
```

Esempio 2

L'esempio seguente è un gestore di richieste di funzioni per una mutazione `updateItem(id: ID!, title: String, author: String, expectedVersion: Int!) GraphQL`.

Si tratta di un esempio complesso che verifica gli argomenti e genera dinamicamente l'espressione di aggiornamento in cui sono inclusi solo gli argomenti forniti dal client. Ad esempio, se `title` e `author` vengono omessi, non vengono aggiornati. Se viene specificato un argomento ma il suo valore è `null`, quel campo viene eliminato dall'oggetto in DynamoDB. Infine, l'operazione ha una condizione, che verifica se l'elemento attualmente in DynamoDB ha `version` il campo impostato su: `expectedVersion`

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { args: { input: { id, ...values } } } = ctx;

  const condition = {
    id: { attributeExists: true },
    version: { eq: values.expectedVersion },
  };
  values.expectedVersion += 1;
  return dynamodbUpdateRequest({ keys: { id }, values, condition });
}

/**
 * Helper function to update an item
 * @returns an UpdateItem request
 */
function dynamodbUpdateRequest(params) {
  const { keys, values, condition: inCondObj } = params;

  const sets = [];
  const removes = [];
  const expressionNames = {};
  const expValues = {};

  // Iterate through the keys of the values
  for (const [key, value] of Object.entries(values)) {
    expressionNames[`#${key}`] = key;
    if (value) {
      sets.push(`#${key} = :${key}`);
    }
  }
}
```

```

    expValues[`${key}`] = value;
  } else {
    removes.push(`${key}`);
  }
}

let expression = sets.length ? `SET ${sets.join(', ')}` : '';
expression += removes.length ? ` REMOVE ${removes.join(', ')}` : '';

const condition = JSON.parse(
  util.transform.toDynamoDBConditionExpression(inCondObj)
);

return {
  operation: 'UpdateItem',
  key: util.dynamodb.toMapValues(keys),
  condition,
  update: {
    expression,
    expressionNames,
    expressionValues: util.dynamodb.toMapValues(expValues),
  },
};
}

```

Per ulteriori informazioni sull'API UpdateItem di DynamoDB, consulta la [documentazione API di DynamoDB](#).

DeleteItem

La DeleteItem richiesta consente di indicare alla funzione AWS AppSync DynamoDB di effettuare una DeleteItem richiesta a DynamoDB e consente di specificare quanto segue:

- La chiave dell'elemento in DynamoDB
- Condizioni per la riuscita dell'operazione

La DeleteItem richiesta ha la seguente struttura:

```

type DynamoDBDeleteItemRequest = {
  operation: 'DeleteItem';
  key: { [key: string]: any };
}

```

```
condition?: ConditionCheckExpression;  
customPartitionKey?: string;  
populateIndexFields?: boolean;  
_version?: number;  
};
```

I campi sono definiti come segue:

DeleteItem campi

DeleteItem elenco dei campi

operation

L'operazione DynamoDB da eseguire. Per eseguire l'operazione DeleteItem DynamoDB, il valore deve essere impostato su DeleteItem. Questo valore è obbligatorio.

key

La chiave dell'elemento in DynamoDB. Gli elementi DynamoDB possono avere una sola chiave hash o una chiave hash e una chiave di ordinamento, a seconda della struttura della tabella.

[Per ulteriori informazioni sulla specificazione di un «valore digitato», consulta Sistema di tipi \(mappatura delle richieste\)](#). Questo valore è obbligatorio.

condition

Una condizione per determinare se la richiesta deve riuscire o no in base allo stato dell'oggetto già incluso in DynamoDB. Se non viene specificata alcuna condizione, la richiesta DeleteItem elimina la voce indipendentemente dal suo stato attuale. [Per ulteriori informazioni sulle condizioni, vedere Espressioni di condizione](#). Questo valore è facoltativo.

_version

Valore numerico che rappresenta l'ultima versione nota di un elemento. Questo valore è facoltativo. Questo campo viene utilizzato per il rilevamento dei conflitti ed è supportato solo nelle origini dati con versione.

customPartitionKey

Se abilitato, questo valore di stringa modifica il formato dei ds_pk record ds_sk and utilizzati dalla tabella delta sync quando il controllo delle versioni è abilitato (per ulteriori informazioni, consulta [Conflict detection and sync](#) nella AWS AppSync Developer Guide). Se abilitata, è abilitata anche l'elaborazione della populateIndexFields voce. Questo campo è facoltativo.

populateIndexFields

Un valore booleano che, se abilitato insieme a `customPartitionKey`, crea nuove voci per ogni record nella tabella delta sync, in particolare nelle colonne `gsi_ds_pk` and `gsi_ds_sk`. Per ulteriori informazioni, consulta [Rilevamento e sincronizzazione dei conflitti](#) nella Guida per gli AWS AppSync sviluppatori. Questo campo è facoltativo.

L'elemento eliminato da DynamoDB viene automaticamente convertito in tipi primitivi GraphQL e JSON ed è disponibile nel contesto `result()`. `context.result`

Per ulteriori informazioni sulla conversione dei tipi in DynamoDB, [vedere Sistema dei tipi](#) (mappatura delle risposte).

[Per ulteriori informazioni sui JavaScript resolver, consulta la panoramica dei resolver. JavaScript](#)

Esempio 1

L'esempio seguente è un gestore di richieste di funzioni per una mutazione `deleteItem(id: ID!)` GraphQL. Se esiste già una voce con questo ID, viene eliminata.

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  return {
    operation: 'DeleteItem',
    key: util.dynamodb.toMapValues({ id: ctx.args.id }),
  };
}
```

Esempio 2

L'esempio seguente è un gestore di richieste di funzioni per una mutazione `deleteItem(id: ID!, expectedVersion: Int!)` GraphQL. Se esiste già una voce con questo ID, viene eliminata, ma solo se il relativo campo `version` è impostato su `expectedVersion`:

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { id, expectedVersion } = ctx.args;
  const condition = {
    id: { attributeExists: true },
    version: { eq: expectedVersion },
  };
}
```

```
};  
return {  
  operation: 'DeleteItem',  
  key: util.dynamodb.toMapValues({ id }),  
  condition: util.transform.toDynamoDBConditionExpression(condition),  
};  
}
```

Per ulteriori informazioni sull'API DeleteItem di DynamoDB, consulta la [documentazione API di DynamoDB](#).

Query

L'oggetto Query request consente di indicare al resolver AWS AppSync DynamoDB di effettuare una Query richiesta a DynamoDB e consente di specificare quanto segue:

- Espressione chiave
- Indice da utilizzare
- Eventuali filtri aggiuntivi
- Numero di voci da restituire
- Se utilizzare letture consistenti
- Direzione della query (avanti o indietro)
- Token di paginazione

L'oggetto della Query richiesta ha la seguente struttura:

```
type DynamoDBQueryRequest = {  
  operation: 'Query';  
  query: {  
    expression: string;  
    expressionNames?: { [key: string]: string };  
    expressionValues?: { [key: string]: any };  
  };  
  index?: string;  
  nextToken?: string;  
  limit?: number;  
  scanIndexForward?: boolean;  
  consistentRead?: boolean;  
  select?: 'ALL_ATTRIBUTES' | 'ALL_PROJECTED_ATTRIBUTES' | 'SPECIFIC_ATTRIBUTES';  
}
```

```
filter?: {  
  expression: string;  
  expressionNames?: { [key: string]: string };  
  expressionValues?: { [key: string]: any };  
};  
projection?: {  
  expression: string;  
  expressionNames?: { [key: string]: string };  
};  
};
```

I campi sono definiti come segue:

Campi di interrogazione

Elenco dei campi di interrogazione

operation

L'operazione DynamoDB da eseguire. Per eseguire l'operazione Query DynamoDB, il valore deve essere impostato su Query. Questo valore è obbligatorio.

query

La query sezione consente di specificare un'espressione di condizione chiave che descrive quali elementi recuperare da DynamoDB. Per ulteriori informazioni su come scrivere espressioni di condizioni chiave, consulta la documentazione di [KeyConditions DynamoDB](#). Questa sezione deve essere specificata.

expression

L'espressione della query. Questo campo deve essere specificato.

expressionNames

Le sostituzioni per i segnaposto dell'attributo di espressione name sotto forma di coppie chiave-valore. La chiave corrisponde a un segnaposto per il nome utilizzato in `expression`, e il valore deve essere una stringa corrispondente al nome dell'attributo dell'elemento in DynamoDB. Questo è un campo facoltativo in cui vanno riportate solo le sostituzioni per i segnaposto dell'attributo di espressione name utilizzate in `expression`.

expressionValues

Le sostituzioni per i segnaposto dell'attributo di espressione value sotto forma di coppie chiave-valore. La chiave corrisponde a un segnaposto per un valore utilizzato in `expression`,

mentre il valore deve essere un valore tipizzato. Per ulteriori informazioni su come specificare un «valore digitato», consulta [Type system](#) (request mapping). Questo valore è obbligatorio. Questo è un campo facoltativo in cui vanno riportate solo le sostituzioni per i segnaposto dell'attributo di espressione value utilizzate in `expression`.

filter

Un ulteriore filtro che può essere utilizzato per filtrare i risultati da DynamoDB prima che siano restituiti. Per ulteriori informazioni sui filtri, consulta [Filtri](#). Questo campo è facoltativo.

index

Nome dell'indice su cui eseguire una query. L'operazione di interrogazione DynamoDB consente di eseguire la scansione degli indici secondari locali e degli indici secondari globali oltre all'indice della chiave primaria alla ricerca di una chiave hash. Se specificato, indica a DynamoDB di interrogare l'indice specificato. Se omesso, la query viene eseguita sull'indice primario della chiave.

nextToken

Il token di paginazione utilizzato per continuare una query precedente, dalla quale deve essere ottenuto. Questo campo è facoltativo.

limit

Il numero massimo di item da valutare (non necessariamente il numero di item corrispondenti). Questo campo è facoltativo.

scanIndexForward

Un valore booleano che indica se la query deve essere eseguita in avanti o indietro. Si tratta di un campo facoltativo, impostato di default su `true`.

consistentRead

Un booleano che indica se utilizzare letture coerenti quando si esegue una query su DynamoDB. Si tratta di un campo facoltativo, impostato di default su `false`.

select

Per impostazione predefinita, il AWS AppSync resolver DynamoDB restituisce solo gli attributi proiettati nell'indice. Se sono necessari più attributi, puoi impostare questo campo. Questo campo è facoltativo. I valori supportati sono:

ALL_ATTRIBUTES

Restituisce tutti gli attributi della voce nella tabella o nell'indice specificati. Se si esegue una query su un indice secondario locale, DynamoDB recupera l'intero elemento dalla tabella principale per ogni elemento corrispondente nell'indice. Se l'indice è configurato per proiettare tutti gli attributi della voce, è possibile ottenere tutti i dati dall'indice secondario locale, senza necessità di recupero.

ALL_PROJECTED_ATTRIBUTES

Consentito solo durante l'esecuzione di una query su un indice. Recupera tutti gli attributi proiettati nell'indice. Se la configurazione dell'indice prevede che vi siano proiettati tutti gli attributi, il valore che restituisce è uguale a quello dato da ALL_ATTRIBUTES.

SPECIFIC_ATTRIBUTES

Restituisce solo gli attributi elencati negli. `projection expression` Questo valore restituito equivale a specificare i `projection expression` senza specificare alcun valore per.

Select

projection

Una proiezione utilizzata per specificare gli attributi da restituire dall'operazione DynamoDB. [Per ulteriori informazioni sulle proiezioni, vedere Proiezioni](#). Questo campo è facoltativo.

I risultati di DynamoDB vengono convertiti automaticamente in tipi primitivi GraphQL e JSON e sono disponibili nel contesto `result()`. `context.result`

Per ulteriori informazioni sulla conversione dei tipi in DynamoDB, [vedere Sistema dei tipi](#) (mappatura delle risposte).

[Per ulteriori informazioni sui JavaScript resolver, consulta la panoramica dei resolver. JavaScript](#)

I risultati hanno la struttura seguente:

```
{
  items = [ ... ],
  nextToken = "a pagination token",
  scannedCount = 10
}
```

I campi sono definiti come segue:

items

Un elenco contenente gli elementi restituiti dalla query DynamoDB.

nextToken

Se è possibile che ci siano altri risultati, `nextToken` contiene un token di paginazione utilizzabile in un'altra richiesta. Nota che AWS AppSync crittografa e offusca il token di impaginazione restituito da DynamoDB. In questo modo si evita che i dati della tabella siano inavvertitamente divulgati all'intermediario. Si noti inoltre che questi token di impaginazione non possono essere utilizzati con funzioni o resolver diversi.

scannedCount

Il numero di voci corrispondenti all'espressione di condizione della query prima dell'applicazione di un'espressione di filtro (se presente).

Esempio

L'esempio seguente è un gestore di richieste di funzioni per una query `getPosts(owner: ID!)` GraphQL.

In questo esempio, un indice secondario globale in una tabella viene interrogato per restituire tutti i post di proprietà dell'ID specificato.

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { owner } = ctx.args;
  return {
    operation: 'Query',
    query: {
      expression: 'ownerId = :ownerId',
      expressionValues: util.dynamodb.toMapValues({ ':ownerId': owner }),
    },
    index: 'owner-index',
  };
}
```

Per ulteriori informazioni sull'API Query di DynamoDB, consulta la [documentazione API di DynamoDB](#).

Scan

La Scan richiesta consente di indicare alla funzione AWS AppSync DynamoDB di effettuare una Scan richiesta a DynamoDB e consente di specificare quanto segue:

- Un filtro per escludere i risultati
- Indice da utilizzare
- Numero di voci da restituire
- Se utilizzare letture consistenti
- Token di paginazione
- Scansioni parallele

L'oggetto della Scan richiesta ha la seguente struttura:

```
type DynamoDBScanRequest = {  
  operation: 'Scan';  
  index?: string;  
  limit?: number;  
  consistentRead?: boolean;  
  nextToken?: string;  
  totalSegments?: number;  
  segment?: number;  
  filter?: {  
    expression: string;  
    expressionNames?: { [key: string]: string };  
    expressionValues?: { [key: string]: any };  
  };  
  projection?: {  
    expression: string;  
    expressionNames?: { [key: string]: string };  
  };  
};
```

I campi sono definiti come segue:

Campi di scansione

Elenco dei campi di scansione

operation

L'operazione DynamoDB da eseguire. Per eseguire l'operazione Scan DynamoDB, il valore deve essere impostato su Scan. Questo valore è obbligatorio.

filter

Un filtro che può essere utilizzato per filtrare i risultati di DynamoDB prima che vengano restituiti. Per ulteriori informazioni sui filtri, consulta [Filtri](#). Questo campo è facoltativo.

index

Nome dell'indice su cui eseguire una query. L'operazione di interrogazione DynamoDB consente di eseguire la scansione degli indici secondari locali e degli indici secondari globali oltre all'indice della chiave primaria alla ricerca di una chiave hash. Se specificato, indica a DynamoDB di interrogare l'indice specificato. Se omissso, la query viene eseguita sull'indice primario della chiave.

limit

Numero massimo di elementi da valutare in una sola volta. Questo campo è facoltativo.

consistentRead

Un valore booleano che indica se utilizzare letture coerenti quando si esegue una query su DynamoDB. Si tratta di un campo facoltativo, impostato di default su `false`.

nextToken

Il token di paginazione utilizzato per continuare una query precedente, dalla quale deve essere ottenuto. Questo campo è facoltativo.

select

Per impostazione predefinita, la funzione AWS AppSync DynamoDB restituisce solo gli attributi proiettati nell'indice. Se sono necessari più attributi, è possibile impostare questo campo. Questo campo è facoltativo. I valori supportati sono:

ALL_ATTRIBUTES

Restituisce tutti gli attributi della voce nella tabella o nell'indice specificati. Se si esegue una query su un indice secondario locale, DynamoDB recupera l'intero elemento dalla tabella

principale per ogni elemento corrispondente nell'indice. Se l'indice è configurato per proiettare tutti gli attributi della voce, è possibile ottenere tutti i dati dall'indice secondario locale, senza necessità di recupero.

ALL_PROJECTED_ATTRIBUTES

Consentito solo durante l'esecuzione di una query su un indice. Recupera tutti gli attributi proiettati nell'indice. Se la configurazione dell'indice prevede che vi siano proiettati tutti gli attributi, il valore che restituisce è uguale a quello dato da ALL_ATTRIBUTES.

SPECIFIC_ATTRIBUTES

Restituisce solo gli attributi elencati negli. `projection expression` Questo valore restituito equivale a specificare i `projection expression` senza specificare alcun valore per. `Select`

totalSegments

Il numero di segmenti in cui partizionare la tabella durante una scansione parallela. Questo campo è facoltativo, ma deve essere specificato se è specificato anche `segment`.

segment

Il segmento della tabella in questa operazione quando si esegue una scansione parallela. Questo campo è facoltativo, ma deve essere specificato se è specificato anche `totalSegments`.

projection

Una proiezione utilizzata per specificare gli attributi da restituire dall'operazione DynamoDB. [Per ulteriori informazioni sulle proiezioni, vedere Proiezioni](#). Questo campo è facoltativo.

I risultati restituiti dalla scansione DynamoDB vengono convertiti automaticamente in tipi primitivi GraphQL e JSON e sono disponibili nel contesto `result()`. `context.result`

Per ulteriori informazioni sulla conversione dei tipi in DynamoDB, [vedere Sistema dei tipi](#) (mappatura delle risposte).

[Per ulteriori informazioni sui JavaScript resolver, consulta la panoramica dei resolver. JavaScript](#)

I risultati hanno la struttura seguente:

```
{
  items = [ ... ],
  nextToken = "a pagination token",
```

```
    scannedCount = 10  
  }
```

I campi sono definiti come segue:

items

Un elenco contenente gli elementi restituiti dalla scansione DynamoDB.

nextToken

Se potrebbero esserci più risultati, `nextToken` contiene un token di impaginazione che puoi utilizzare in un'altra richiesta. AWS AppSync crittografa e offusca il token di impaginazione restituito da DynamoDB. In questo modo si evita che i dati della tabella siano inavvertitamente divulgati all'intermediario. Inoltre, questi token di impaginazione non possono essere utilizzati con funzioni o resolver diversi.

scannedCount

Il numero di elementi recuperati da DynamoDB prima dell'applicazione di un'espressione di filtro (se presente).

Esempio 1

L'esempio seguente è un gestore di richieste di funzioni per la query GraphQL: `allPosts`

In questo esempio, vengono restituite tutte le voci della tabella.

```
export function request(ctx) {  
  return { operation: 'Scan' };  
}
```

Esempio 2

L'esempio seguente è un gestore di richieste di funzioni per la query GraphQL:

`postsMatching(title: String!)`

In questo esempio, vengono restituite tutte le voci della tabella il cui titolo inizia con l'argomento `title`.

```
export function request(ctx) {
```

```
const { title } = ctx.args;
const filter = { filter: { beginsWith: title } };
return {
  operation: 'Scan',
  filter: JSON.parse(util.transform.toDynamoDBFilterExpression(filter)),
};
}
```

Per ulteriori informazioni sull'API Scan di DynamoDB, consulta la [documentazione API di DynamoDB](#).

Sync

L'oggetto di Sync richiesta consente di recuperare tutti i risultati da una tabella DynamoDB e quindi ricevere solo i dati modificati dall'ultima query (gli aggiornamenti delta). Syncle richieste possono essere inoltrate solo a sorgenti dati DynamoDB con versione. È possibile specificare quanto segue:

- Un filtro per escludere i risultati
- Numero di voci da restituire
- Token di paginazione
- Quando è stata avviata l'ultima operazione Sync

L'oggetto della Sync richiesta ha la seguente struttura:

```
type DynamoDBSyncRequest = {
  operation: 'Sync';
  basePartitionKey?: string;
  deltaIndexName?: string;
  limit?: number;
  nextToken?: string;
  lastSync?: number;
  filter?: {
    expression: string;
    expressionNames?: { [key: string]: string };
    expressionValues?: { [key: string]: any };
  };
};
```

I campi sono definiti come segue:

Sincronizza campi

Elenco dei campi di sincronizzazione

operation

L'operazione DynamoDB da eseguire. Per eseguire l'operazione Sync, il valore deve essere impostato su Sync. Questo valore è obbligatorio.

filter

Un filtro che può essere utilizzato per filtrare i risultati di DynamoDB prima che vengano restituiti. Per ulteriori informazioni sui filtri, consulta [Filtri](#). Questo campo è facoltativo.

limit

Numero massimo di elementi da valutare in una sola volta. Questo campo è facoltativo. Se omissso, il limite predefinito sarà impostato su 100 elementi. Il valore massimo per questo campo è 1000 articoli.

nextToken

Il token di paginazione utilizzato per continuare una query precedente, dalla quale deve essere ottenuto. Questo campo è facoltativo.

lastSync

Il momento, in millisecondi dall'epoca, in cui è iniziata l'ultima operazione Sync riuscita. Se specificato, vengono restituiti solo gli elementi che sono stati modificati dopo lastSync. Questo campo è facoltativo e deve essere compilato solo dopo aver recuperato tutte le pagine da un'operazione Sync iniziale. Se omissso, i risultati della tabella Base verranno restituiti, altrimenti verranno restituiti i risultati della tabella Delta.

basePartitionKey

La chiave di partizione della tabella Base utilizzata durante l'esecuzione di un'operazione. Sync Questo campo consente di eseguire un'Syncoperazione quando la tabella utilizza una chiave di partizione personalizzata. Questo campo è opzionale.

deltaIndexName

L'indice utilizzato per l'operazione. Sync Questo indice è necessario per abilitare un'Syncoperazione sull'intera tabella delta store quando la tabella utilizza una chiave di

partizione personalizzata. L'operazione verrà eseguita sul GSI (creato su `gsi_ds_pk` and `gsi_ds_sk`). Questo campo è facoltativo.

I risultati restituiti dalla sincronizzazione DynamoDB vengono convertiti automaticamente in tipi primitivi GraphQL e JSON e sono disponibili nel contesto `result()`. `context.result`

Per ulteriori informazioni sulla conversione dei tipi in DynamoDB, [vedere Sistema dei tipi](#) (mappatura delle risposte).

[Per ulteriori informazioni sui JavaScript resolver, consulta la panoramica dei resolver. JavaScript](#)

I risultati hanno la struttura seguente:

```
{
  items = [ ... ],
  nextToken = "a pagination token",
  scannedCount = 10,
  startedAt = 1550000000000
}
```

I campi sono definiti come segue:

items

Un elenco contenente le voci restituite dalla sincronizzazione.

nextToken

Se potrebbero esserci più risultati, `nextToken` contiene un token di impaginazione che puoi utilizzare in un'altra richiesta. AWS AppSync crittografa e offusca il token di impaginazione restituito da DynamoDB. In questo modo si evita che i dati della tabella siano inavvertitamente divulgati all'intermediario. Inoltre, questi token di impaginazione non possono essere utilizzati con funzioni o resolver diversi.

scannedCount

Il numero di elementi recuperati da DynamoDB prima dell'applicazione di un'espressione di filtro (se presente).

startedAt

Il momento, in millisecondi dall'epoca, in cui è iniziata l'operazione di sincronizzazione che è possibile memorizzare localmente e utilizzare in un'altra richiesta come argomento `lastSync`.

Se un token di paginazione è stato incluso nella richiesta, questo valore sarà lo stesso di quello restituito dalla richiesta per la prima pagina di risultati.

Esempio

L'esempio seguente è un gestore di richieste di funzioni per la query GraphQL:

```
syncPosts(nextToken: String, lastSync: AWSTimestamp)
```

In questo esempio, se `lastSync` viene omissso, vengono restituite tutte le voci nella tabella di base. Se `lastSync` viene fornito, vengono restituite solo le voci nella tabella di sincronizzazione delta che sono state modificate dal momento in cui sono state sincronizzate `lastSync`.

```
export function request(ctx) {  
  const { nextToken, lastSync } = ctx.args;  
  return { operation: 'Sync', limit: 100, nextToken, lastSync };  
}
```

BatchGetItem

L'oggetto `BatchGetItem` request consente di indicare alla funzione AWS AppSync DynamoDB di effettuare una `BatchGetItem` richiesta a DynamoDB per recuperare più elementi, potenzialmente su più tabelle. Per questo oggetto di richiesta, è necessario specificare quanto segue:

- I nomi delle tabelle da cui recuperare le voci
- Le chiavi delle voci da recuperare da ciascuna tabella

Si applicano i limiti `BatchGetItem` di DynamoDB e non si può inserire alcuna espressione di condizione.

L'oggetto della `BatchGetItem` richiesta ha la seguente struttura:

```
type DynamoDBBatchGetItemRequest = {  
  operation: 'BatchGetItem';  
  tables: {  
    [tableName: string]: {  
      keys: { [key: string]: any }[];  
      consistentRead?: boolean;  
      projection?: {
```

```
        expression: string;
        expressionNames?: { [key: string]: string };
    };
};
};
};
```

I campi sono definiti come segue:

BatchGetItem campi

BatchGetItem elenco dei campi

operation

L'operazione DynamoDB da eseguire. Per eseguire l'operazione BatchGetItem DynamoDB, il valore deve essere impostato su BatchGetItem. Questo valore è obbligatorio.

tables

Le tabelle DynamoDB da cui recuperare gli elementi. Il valore è una mappa in cui i nomi delle tabelle sono specificati come chiavi della mappa. Occorre specificare almeno una tabella. Questo valore `tables` è obbligatorio.

keys

Elenco di chiavi DynamoDB che rappresentano la chiave primaria degli elementi da recuperare. Gli elementi DynamoDB possono avere una sola chiave hash o una chiave hash e una chiave di ordinamento, a seconda della struttura della tabella. Per ulteriori informazioni su come specificare un «valore digitato», consulta [Sistema di tipi \(mappatura delle richieste\)](#).

consistentRead

Se utilizzare una lettura coerente durante l'esecuzione di un'GetItemoperazione. Questo valore è opzionale e l'impostazione predefinita è `false`.

projection

Una proiezione utilizzata per specificare gli attributi da restituire dall'operazione DynamoDB. [Per ulteriori informazioni sulle proiezioni, vedere Proiezioni](#). Questo campo è facoltativo.

Aspetti da ricordare:

- Se una voce non è stata recuperata dalla tabella, un elemento null compare nel blocco di dati relativo a quella tabella.
- I risultati delle chiamate vengono ordinati per tabella, in base all'ordine in cui sono stati forniti all'interno dell'oggetto della richiesta.
- Ogni Get comando all'interno di a BatchGetItem è atomico, tuttavia un batch può essere parzialmente elaborato. Se un batch viene elaborato parzialmente a causa di un errore, le chiavi non elaborate vengono restituite nell'ambito del risultato dell'invocazione all'interno del blocco unprocessedKeys.
- BatchGetItem ha un limite di 100 chiavi.

Per il seguente esempio di gestore di richieste di funzioni:

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { authorId, postId } = ctx.args;
  return {
    operation: 'BatchGetItem',
    tables: {
      authors: [util.dynamodb.toMapValues({ authorId })],
      posts: [util.dynamodb.toMapValues({ authorId, postId })],
    },
  };
}
```

Il risultato dell'invocazione disponibile in `ctx.result` è il seguente:

```
{
  "data": {
    "authors": [null],
    "posts": [
      // Was retrieved
      {
        "authorId": "a1",
        "postId": "p2",
        "postTitle": "title",
        "postDescription": "description",
      }
    ]
  },
}
```

```
"unprocessedKeys": {
  "authors": [
    // This item was not processed due to an error
    {
      "authorId": "a1"
    }
  ],
  "posts": []
}
```

Il messaggio `ctx.error` contiene dettagli relativi all'errore. È garantito che i dati delle chiavi, `unprocessedKeys` e ogni chiave di tabella fornita nel risultato nell'oggetto di richiesta della funzione siano presenti nel risultato della chiamata. Le voci eliminate compaiono nel blocco dati. Le voci non elaborate vengono contrassegnate come null all'interno del blocco dati e vengono inserite nel blocco `unprocessedKeys`.

BatchDeleteItem

L'oggetto `BatchDeleteItem request` consente di indicare alla funzione AWS AppSync DynamoDB di effettuare una `BatchWriteItem` richiesta a DynamoDB per eliminare più elementi, potenzialmente su più tabelle. Per questo oggetto di richiesta, è necessario specificare quanto segue:

- I nomi delle tabelle da cui eliminare le voci
- Le chiavi delle voci da eliminare da ciascuna tabella

Si applicano i limiti `BatchWriteItem` di DynamoDB e non si può inserire alcuna espressione di condizione.

L'oggetto della `BatchDeleteItem` richiesta ha la seguente struttura:

```
type DynamoDBBatchDeleteItemRequest = {
  operation: 'BatchDeleteItem';
  tables: {
    [tableName: string]: { [key: string]: any }[];
  };
};
```

I campi sono definiti come segue:

BatchDeleteItem campi

BatchDeleteItem elenco dei campi

operation

L'operazione DynamoDB da eseguire. Per eseguire l'operazione BatchDeleteItem DynamoDB, il valore deve essere impostato su BatchDeleteItem. Questo valore è obbligatorio.

tables

Le tabelle DynamoDB da cui eliminare gli elementi. Ogni tabella è un elenco di chiavi DynamoDB che rappresentano la chiave primaria degli elementi da eliminare. Gli elementi DynamoDB possono avere una sola chiave hash o una chiave hash e una chiave di ordinamento, a seconda della struttura della tabella. Per ulteriori informazioni su come specificare un «valore digitato», consulta [Sistema di tipi \(mappatura delle richieste\)](#). Occorre specificare almeno una tabella. Il tables valore è obbligatorio.

Aspetti da ricordare:

- A differenza dell'operazione DeleteItem, nella risposta non viene restituita la voce completamente eliminata. Viene restituita solo la chiave passata.
- Se una voce non è stata eliminata dalla tabella, un elemento null compare nel blocco di dati relativo a quella tabella.
- I risultati della chiamata vengono ordinati per tabella, in base all'ordine in cui sono stati forniti all'interno dell'oggetto della richiesta.
- Ogni Delete comando all'interno di a è atomicoBatchDeleteItem. Tuttavia, un batch può essere parzialmente elaborato. Se un batch viene elaborato parzialmente a causa di un errore, le chiavi non elaborate vengono restituite nell'ambito del risultato dell'invocazione all'interno del blocco unprocessedKeys.
- BatchDeleteItem ha un limite di 25 chiavi.
- Questa operazione non è supportata se utilizzata con il rilevamento dei conflitti. L'utilizzo di entrambi allo stesso tempo può causare un errore.

Per il seguente esempio di gestore di richieste di funzioni:

```
import { util } from '@aws-appsync/utils';
```

```
export function request(ctx) {
  const { authorId, postId } = ctx.args;
  return {
    operation: 'BatchDeleteItem',
    tables: {
      authors: [util.dynamodb.toMapValues({ authorId })],
      posts: [util.dynamodb.toMapValues({ authorId, postId })],
    },
  };
}
```

Il risultato dell'invocazione disponibile in `ctx.result` è il seguente:

```
{
  "data": {
    "authors": [null],
    "posts": [
      // Was deleted
      {
        "authorId": "a1",
        "postId": "p2"
      }
    ]
  },
  "unprocessedKeys": {
    "authors": [
      // This key was not processed due to an error
      {
        "authorId": "a1"
      }
    ],
    "posts": []
  }
}
```

Il messaggio `ctx.error` contiene dettagli relativi all'errore. È garantito che i dati delle chiavi, `UnprocessedKeys` e ogni chiave di tabella fornita nell'oggetto di richiesta della funzione siano presenti nel risultato dell'invocazione. Le voci eliminate sono presenti nel blocco di dati. Le voci non elaborate vengono contrassegnate come `null` all'interno del blocco dati e vengono inserite nel blocco `unprocessedKeys`.

BatchPutItem

L'oggetto `BatchPutItem` request consente di indicare alla funzione AWS AppSync DynamoDB di effettuare una `BatchWriteItem` richiesta a DynamoDB per inserire più elementi, potenzialmente su più tabelle. Per questo oggetto di richiesta, è necessario specificare quanto segue:

- I nomi delle tabelle in cui inserire le voci
- Le voci complete da inserire in ciascuna tabella

Si applicano i limiti `BatchWriteItem` di DynamoDB e non si può inserire alcuna espressione di condizione.

L'oggetto della `BatchPutItem` richiesta ha la seguente struttura:

```
type DynamoDBBatchPutItemRequest = {  
  operation: 'BatchPutItem';  
  tables: {  
    [tableName: string]: { [key: string]: any }[];  
  };  
};
```

I campi sono definiti come segue:

BatchPutItem campi

BatchPutItem elenco dei campi

operation

L'operazione DynamoDB da eseguire. Per eseguire l'operazione `BatchPutItem` DynamoDB, il valore deve essere impostato su `BatchPutItem`. Questo valore è obbligatorio.

tables

Le tabelle DynamoDB in cui inserire gli elementi. Ogni voce della tabella rappresenta un elenco di elementi DynamoDB da inserire per questa tabella specifica. Occorre specificare almeno una tabella. Questo valore è obbligatorio.

Aspetti da ricordare:

- Se l'operazione va a buon fine, nella risposta vengono restituite le voci inserite completamente.
- Se una voce non è stata inserita nella tabella, un elemento null viene visualizzato nel blocco di dati relativo a quella tabella.
- Gli elementi inseriti vengono ordinati per tabella, in base all'ordine in cui sono stati forniti all'interno dell'oggetto della richiesta.
- Ogni Put comando all'interno di a BatchPutItem è atomico, tuttavia un batch può essere parzialmente elaborato. Se un batch viene elaborato parzialmente a causa di un errore, le chiavi non elaborate vengono restituite nell'ambito del risultato dell'invocazione all'interno del blocco unprocessedKeys.
- BatchPutItem ha un limite di 25 voci.
- Questa operazione non è supportata se utilizzata con il rilevamento dei conflitti. L'utilizzo di entrambi allo stesso tempo può causare un errore.

Per il seguente esempio di gestore di richieste di funzioni:

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { authorId, postId, name, title } = ctx.args;
  return {
    operation: 'BatchPutItem',
    tables: {
      authors: [util.dynamodb.toMapValues({ authorId, name })],
      posts: [util.dynamodb.toMapValues({ authorId, postId, title })],
    },
  };
}
```

Il risultato dell'invocazione disponibile in `ctx.result` è il seguente:

```
{
  "data": {
    "authors": [
      null
    ],
    "posts": [
      // Was inserted
      {
        "authorId": "a1",
```

```

        "postId": "p2",
        "title": "title"
      }
    ]
  },
  "unprocessedItems": {
    "authors": [
      // This item was not processed due to an error
      {
        "authorId": "a1",
        "name": "a1_name"
      }
    ],
    "posts": []
  }
}

```

Il messaggio `ctx.error` contiene dettagli relativi all'errore. È garantito che i dati delle chiavi, `UnprocessedItems` e ogni chiave di tabella fornita nell'oggetto di richiesta siano presenti nel risultato della chiamata. Le voci inserite si trovano nel blocco di dati. Le voci non elaborate vengono contrassegnate come null all'interno del blocco dati e vengono inserite nel blocco `unprocessedItems`.

TransactGetItems

L'oggetto `TransactGetItems request` consente di indicare alla funzione AWS AppSync DynamoDB di effettuare una `TransactGetItems` richiesta a DynamoDB per recuperare più elementi, potenzialmente su più tabelle. Per questo oggetto di richiesta, è necessario specificare quanto segue:

- Il nome della tabella di ogni elemento di richiesta da cui recuperare l'elemento
- La chiave di ogni elemento di richiesta da recuperare da ogni tabella

Si applicano i limiti `TransactGetItems` di DynamoDB e non si può inserire alcuna espressione di condizione.

L'oggetto della `TransactGetItems` richiesta ha la seguente struttura:

```

type DynamoDBTransactGetItemsRequest = {
  operation: 'TransactGetItems';
  transactItems: { table: string; key: { [key: string]: any }; projection?:
{ expression: string; expressionNames?: { [key: string]: string }; }[];
};

```

```
};
```

I campi sono definiti come segue:

TransactGetItems campi

TransactGetItems elenco dei campi

operation

L'operazione DynamoDB da eseguire. Per eseguire l'operazione TransactGetItems DynamoDB, il valore deve essere impostato su TransactGetItems. Questo valore è obbligatorio.

transactItems

Gli elementi di richiesta da includere. Il valore è un array di elementi di richiesta. Deve essere fornito almeno un elemento di richiesta. Questo valore transactItems è obbligatorio.

table

La tabella DynamoDB da cui recuperare l'elemento. Il valore è una stringa del nome della tabella. Questo valore table è obbligatorio.

key

La chiave DynamoDB che rappresenta la chiave primaria dell'elemento da recuperare. Gli elementi DynamoDB possono avere una sola chiave hash o una chiave hash e una chiave di ordinamento, a seconda della struttura della tabella. Per ulteriori informazioni su come specificare un «valore digitato», consulta [Sistema di tipi \(mappatura delle richieste\)](#).

projection

Una proiezione utilizzata per specificare gli attributi da restituire dall'operazione DynamoDB. [Per ulteriori informazioni sulle proiezioni, vedere Proiezioni](#). Questo campo è facoltativo.

Aspetti da ricordare:

- Se una transazione ha esito positivo, l'ordine degli elementi recuperati nel blocco items sarà lo stesso dell'ordine degli elementi della richiesta.
- Le transazioni vengono eseguite in qualche modo. all-or-nothing Se un elemento di richiesta causa un errore, l'intera transazione non viene eseguita e vengono restituiti i dettagli dell'errore.

- Un elemento di richiesta che non può essere recuperato non è un errore. Invece, un elemento null appare nel blocco elementi nella posizione corrispondente.
- Se l'errore di una transazione è `TransactionCanceledException`, il `cancellationReasons` blocco verrà popolato. L'ordine dei motivi di annullamento nel blocco `cancellationReasons` è lo stesso dell'ordine degli elementi della richiesta.
- `TransactGetItems` è limitato a 100 articoli richiesti.

Per il seguente esempio di gestore di richieste di funzioni:

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { authorId, postId } = ctx.args;
  return {
    operation: 'TransactGetItems',
    transactItems: [
      {
        table: 'posts',
        key: util.dynamodb.toMapValues({ postId }),
      },
      {
        table: 'authors',
        key: util.dynamodb.toMapValues({ authorId }),
      },
    ],
  };
}
```

Se la transazione ha esito positivo e viene recuperato solo il primo elemento richiesto, il risultato della chiamata disponibile `ctx.result` è il seguente:

```
{
  "items": [
    {
      // Attributes of the first requested item
      "post_id": "p1",
      "post_title": "title",
      "post_description": "description"
    },
    // Could not retrieve the second requested item
    null,
  ]
}
```

```
],
  "cancellationReasons": null
}
```

Se la transazione fallisce a `TransactionCanceledException` causata dal primo elemento della richiesta, il risultato della chiamata disponibile in `ctx.result` è il seguente:

```
{
  "items": null,
  "cancellationReasons": [
    {
      "type": "Sample error type",
      "message": "Sample error message"
    },
    {
      "type": "None",
      "message": "None"
    }
  ]
}
```

Il messaggio `ctx.error` contiene dettagli relativi all'errore. La presenza delle chiavi `items` e `cancellationReasons` è garantita in `ctx.result`.

TransactWriteItems

L'oggetto `TransactWriteItems request` consente di indicare alla funzione AWS AppSync DynamoDB di `TransactWriteItems` richiedere a DynamoDB di scrivere più elementi, potenzialmente su più tabelle. Per questo oggetto di richiesta, è necessario specificare quanto segue:

- Il nome della tabella di destinazione di ogni elemento di richiesta
- L'operazione di ogni elemento di richiesta da eseguire. Sono supportati quattro tipi di operazioni: `PutItem`, `UpdateItem`, `DeleteItem`, e `ConditionCheck`
- La chiave di ogni elemento di richiesta da scrivere

Si applicano i limiti `TransactWriteItems` DynamoDB.

L'oggetto della `TransactWriteItems` richiesta ha la seguente struttura:

```
type DynamoDBTransactWriteItemsRequest = {
```

```

    operation: 'TransactWriteItems';
    transactItems: TransactItem[];
  };
  type TransactItem =
    | TransactWritePutItem
    | TransactWriteUpdateItem
    | TransactWriteDeleteItem
    | TransactWriteConditionCheckItem;
  type TransactWritePutItem = {
    table: string;
    operation: 'PutItem';
    key: { [key: string]: any };
    attributeValues: { [key: string]: string };
    condition?: TransactConditionCheckExpression;
  };
  type TransactWriteUpdateItem = {
    table: string;
    operation: 'UpdateItem';
    key: { [key: string]: any };
    update: DynamoDBExpression;
    condition?: TransactConditionCheckExpression;
  };
  type TransactWriteDeleteItem = {
    table: string;
    operation: 'DeleteItem';
    key: { [key: string]: any };
    condition?: TransactConditionCheckExpression;
  };
  type TransactWriteConditionCheckItem = {
    table: string;
    operation: 'ConditionCheck';
    key: { [key: string]: any };
    condition?: TransactConditionCheckExpression;
  };
  type TransactConditionCheckExpression = {
    expression: string;
    expressionNames?: { [key: string]: string };
    expressionValues?: { [key: string]: any };
    returnValuesOnConditionCheckFailure: boolean;
  };
};

```

TransactWriteItems campi

TransactWriteItems elenco dei campi

I campi sono definiti come segue:

operation

L'operazione DynamoDB da eseguire. Per eseguire l'operazione TransactWriteItems DynamoDB, il valore deve essere impostato su TransactWriteItems. Questo valore è obbligatorio.

transactItems

Gli elementi di richiesta da includere. Il valore è un array di elementi di richiesta. Deve essere fornito almeno un elemento di richiesta. Questo valore transactItems è obbligatorio.

Per PutItem, i campi sono definiti come segue:

table

La tabella DynamoDB di destinazione. Il valore è una stringa del nome della tabella. Questo valore table è obbligatorio.

operation

L'operazione DynamoDB da eseguire. Per eseguire l'operazione PutItem DynamoDB, il valore deve essere impostato su PutItem. Questo valore è obbligatorio.

key

La chiave DynamoDB che rappresenta la chiave primaria dell'elemento da inserire. Gli elementi DynamoDB possono avere una sola chiave hash o una chiave hash e una chiave di ordinamento, a seconda della struttura della tabella. Per ulteriori informazioni su come specificare un «valore digitato», consulta [Sistema di tipi \(mappatura delle richieste\)](#). Questo valore è obbligatorio.

attributeValues

Gli altri attributi della voce da inserir in DynamoDB. Per ulteriori informazioni su come specificare un «valore digitato», vedete [Type system \(request mapping\)](#). Questo campo è facoltativo.

condition

Una condizione per determinare se la richiesta deve riuscire o no in base allo stato dell'oggetto già incluso in DynamoDB. Se non viene specificata alcuna condizione, la richiesta PutItem sovrascrive qualsiasi valore esistente per quella voce. È possibile specificare se recuperare l'elemento esistente quando il controllo delle condizioni non riesce. [Per ulteriori informazioni sulle condizioni transazionali, consulta Espressioni delle condizioni di transazione](#). Questo valore è facoltativo.

Per UpdateItem, i campi sono definiti come segue:

table

La tabella DynamoDB da aggiornare. Il valore è una stringa del nome della tabella. Questo valore table è obbligatorio.

operation

L'operazione DynamoDB da eseguire. Per eseguire l'operazione UpdateItem DynamoDB, il valore deve essere impostato su UpdateItem. Questo valore è obbligatorio.

key

La chiave DynamoDB che rappresenta la chiave primaria dell'elemento da aggiornare. Gli elementi DynamoDB possono avere una sola chiave hash o una chiave hash e una chiave di ordinamento, a seconda della struttura della tabella. Per ulteriori informazioni su come specificare un «valore digitato», consulta [Sistema di tipi \(mappatura delle richieste\)](#). Questo valore è obbligatorio.

update

La update sezione consente di specificare un'espressione di aggiornamento che descrive come aggiornare l'elemento in DynamoDB. Per ulteriori informazioni su come scrivere espressioni di aggiornamento, consulta la documentazione di [UpdateExpressions DynamoDB](#). Questa sezione è obbligatoria.

condition

Una condizione per determinare se la richiesta deve riuscire o no in base allo stato dell'oggetto già incluso in DynamoDB. Se non viene specificata alcuna condizione, la richiesta UpdateItem aggiorna qualsiasi valore esistente, indipendentemente dal suo stato attuale. È possibile specificare se recuperare l'elemento esistente quando il controllo

delle condizioni non riesce. [Per ulteriori informazioni sulle condizioni transazionali, consulta Espressioni delle condizioni di transazione](#). Questo valore è facoltativo.

Per DeleteItem, i campi sono definiti come segue:

table

La tabella DynamoDB in cui eliminare l'elemento. Il valore è una stringa del nome della tabella. Questo valore `table` è obbligatorio.

operation

L'operazione DynamoDB da eseguire. Per eseguire l'operazione DeleteItem DynamoDB, il valore deve essere impostato su DeleteItem. Questo valore è obbligatorio.

key

La chiave DynamoDB che rappresenta la chiave primaria dell'elemento da eliminare. Gli elementi DynamoDB possono avere una sola chiave hash o una chiave hash e una chiave di ordinamento, a seconda della struttura della tabella. Per ulteriori informazioni su come specificare un «valore digitato», consulta [Sistema di tipi \(mappatura delle richieste\)](#). Questo valore è obbligatorio.

condition

Una condizione per determinare se la richiesta deve riuscire o no in base allo stato dell'oggetto già incluso in DynamoDB. Se non viene specificata alcuna condizione, la richiesta DeleteItem elimina la voce indipendentemente dal suo stato attuale. È possibile specificare se recuperare l'elemento esistente quando il controllo delle condizioni non riesce. [Per ulteriori informazioni sulle condizioni transazionali, consulta Espressioni delle condizioni di transazione](#). Questo valore è facoltativo.

Per ConditionCheck, i campi sono definiti come segue:

table

La tabella DynamoDB in cui controllare la condizione. Il valore è una stringa del nome della tabella. Questo valore `table` è obbligatorio.

operation

L'operazione DynamoDB da eseguire. Per eseguire l'operazione ConditionCheck DynamoDB, il valore deve essere impostato su ConditionCheck. Questo valore è obbligatorio.

key

La chiave DynamoDB che rappresenta la chiave primaria dell'elemento per il controllo delle condizioni. Gli elementi DynamoDB possono avere una sola chiave hash o una chiave hash e una chiave di ordinamento, a seconda della struttura della tabella. Per ulteriori informazioni su come specificare un «valore digitato», consulta [Sistema di tipi \(mappatura delle richieste\)](#). Questo valore è obbligatorio.

condition

Una condizione per determinare se la richiesta deve riuscire o no in base allo stato dell'oggetto già incluso in DynamoDB. È possibile specificare se recuperare l'elemento esistente quando il controllo delle condizioni non riesce. [Per ulteriori informazioni sulle condizioni transazionali, consulta Espressioni delle condizioni di transazione.](#) Questo valore è obbligatorio.

Aspetti da ricordare:

- Solo le chiavi degli elementi della richiesta vengono restituite nella risposta, in caso di esito positivo. L'ordine delle chiavi sarà lo stesso dell'ordine degli elementi della richiesta.
- Le transazioni vengono eseguite in qualsiasi modo. all-or-nothing Se un elemento di richiesta causa un errore, l'intera transazione non viene eseguita e vengono restituiti i dettagli dell'errore.
- Nessun elemento di richiesta può scegliere come target lo stesso elemento. Altrimenti causeranno `TransactionCanceledException` errori.
- Se l'errore di una transazione è `TransactionCanceledException`, il `cancellationReasons` blocco verrà popolato. Se il controllo delle condizioni di un elemento di richiesta fallisce e non è stato specificato `returnValuesOnConditionCheckFailure` come `false`, l'elemento esistente nella tabella viene recuperato e memorizzato in `item` nella posizione corrispondente del blocco `cancellationReasons`.
- `TransactWriteItems` è limitato a 100 articoli richiesti.
- Questa operazione non è supportata se utilizzata con il rilevamento dei conflitti. L'utilizzo di entrambi allo stesso tempo può causare un errore.

Per il seguente esempio di gestore di richieste di funzioni:

```
import { util } from '@aws-appsync/utils';
```

```

export function request(ctx) {
  const { authorId, postId, title, description, oldTitle, authorName } = ctx.args;
  return {
    operation: 'TransactWriteItems',
    transactItems: [
      {
        table: 'posts',
        operation: 'PutItem',
        key: util.dynamodb.toMapValues({ postId }),
        attributeValues: util.dynamodb.toMapValues({ title, description }),
        condition: util.transform.toDynamoDBConditionExpression({
          title: { eq: oldTitle },
        }),
      },
      {
        table: 'authors',
        operation: 'UpdateItem',
        key: util.dynamodb.toMapValues({ authorId }),
        update: {
          expression: 'SET authorName = :name',
          expressionValues: util.dynamodb.toMapValues({ ':name': authorName }),
        },
      },
    ],
  };
}

```

Se la transazione ha esito positivo, il risultato della chiamata disponibile in `ctx.result` è il seguente:

```

{
  "keys": [
    // Key of the PutItem request
    {
      "post_id": "p1",
    },
    // Key of the UpdateItem request
    {
      "author_id": "a1"
    }
  ],
  "cancellationReasons": null
}

```

Se la transazione fallisce a causa del fallimento del controllo delle condizioni della `PutItem` richiesta, il risultato della chiamata disponibile in `ctx.result` è il seguente:

```
{
  "keys": null,
  "cancellationReasons": [
    {
      "item": {
        "post_id": "p1",
        "post_title": "Actual old title",
        "post_description": "Old description"
      },
      "type": "ConditionCheckFailed",
      "message": "The condition check failed."
    },
    {
      "type": "None",
      "message": "None"
    }
  ]
}
```

Il messaggio `ctx.error` contiene dettagli relativi all'errore. La presenza delle chiavi `keys` e `cancellationReasons` è garantita in `ctx.result`.

Sistema di tipi (mappatura delle richieste)

Quando si utilizza la funzione AWS AppSync DynamoDB per chiamare le tabelle DynamoDB AWS AppSync, è necessario conoscere il tipo di ogni valore da utilizzare in quella chiamata. Questo perché DynamoDB supporta più primitive di tipo rispetto a GraphQL o JSON (come set e dati binari). AWS AppSync necessita di alcuni suggerimenti per la traduzione tra GraphQL e DynamoDB, altrimenti dovrebbe formulare alcune ipotesi su come i dati sono strutturati nella tabella.

[Per ulteriori informazioni sui tipi di dati DynamoDB, consulta i descrittori dei tipi di dati e la documentazione sui tipi di dati di DynamoDB.](#)

Un valore DynamoDB è rappresentato da un oggetto JSON contenente una singola coppia chiave-valore. La chiave specifica il tipo DynamoDB e il valore specifica il valore stesso. In questo esempio, la chiave `S` indica che il valore è una stringa e il valore `identifier` è il valore della stringa stessa.

```
{ "S" : "identifier" }
```

L'oggetto JSON non può contenere più di una coppia chiave-valore. Se viene specificata più di una coppia chiave-valore, l'oggetto della richiesta non viene analizzato.

Un valore DynamoDB viene utilizzato ovunque in un oggetto di richiesta in cui è necessario specificare un valore. ad esempio nelle sezioni `key` e `attributeValue`, nonché nella sezione `expressionValues` delle sezioni delle espressioni. Nell'esempio seguente, il `identifier` valore DynamoDB String viene assegnato al `id` campo in `key` una sezione (magari in `GetItem` un oggetto di richiesta).

```
"key" : {  
  "id" : { "S" : "identifier" }  
}
```

Tipi supportati

AWS AppSync supporta i seguenti tipi di scalari, documenti e set di DynamoDB:

S (tipo String)

Un singolo valore di stringa. Un valore di stringa DynamoDB è indicato da:

```
{ "S" : "some string" }
```

Un esempio di utilizzo è:

```
"key" : {  
  "id" : { "S" : "some string" }  
}
```

SS (tipo String set)

Un set di valori di stringa. Un valore DynamoDB String Set è indicato da:

```
{ "SS" : [ "first value", "second value", ... ] }
```

Un esempio di utilizzo è:

```
"attributeValues" : {  
  "phoneNumbers" : { "SS" : [ "+1 555 123 4567", "+1 555 234 5678" ] }  
}
```

N (tipo Number)

Un singolo valore numerico. Un valore numerico di DynamoDB è indicato da:

```
{ "N" : 1234 }
```

Un esempio di utilizzo è:

```
"expressionValues" : {  
  ":expectedVersion" : { "N" : 1 }  
}
```

NS (tipo Number set)

Un set di valori numerici. Un valore del DynamoDB Number Set è indicato da:

```
{ "NS" : [ 1, 2.3, 4 ... ] }
```

Un esempio di utilizzo è:

```
"attributeValues" : {  
  "sensorReadings" : { "NS" : [ 67.8, 12.2, 70 ] }  
}
```

B (tipo Binary)

Un valore binario. Un valore binario di DynamoDB è indicato da:

```
{ "B" : "SGVsbG8sIFdvcmxkIQo=" }
```

Nota che il valore è in realtà una stringa, dove la stringa è la rappresentazione codificata in base64 dei dati binari. AWS AppSync decodifica nuovamente questa stringa nel suo valore binario prima di inviarla a DynamoDB. AWS AppSync utilizza lo schema di decodifica base64 come definito da RFC 2045: qualsiasi carattere che non sia nell'alfabeto base64 viene ignorato.

Un esempio di utilizzo è:

```
"attributeValues" : {  
  "binaryMessage" : { "B" : "SGVsbG8sIFdvcmxkIQo=" }  
}
```

```
}
```

BS (tipo Binary set)

Un set di valori binari. Un valore del set binario di DynamoDB è indicato da:

```
{ "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ] }
```

Nota che il valore è in realtà una stringa, dove la stringa è la rappresentazione codificata in base64 dei dati binari. AWS AppSync decodifica nuovamente questa stringa nel suo valore binario prima di inviarla a DynamoDB. AWS AppSync utilizza lo schema di decodifica base64 come definito da RFC 2045: qualsiasi carattere che non sia nell'alfabeto base64 viene ignorato.

Un esempio di utilizzo è:

```
"attributeValues" : {  
  "binaryMessages" : { "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ] }  
}
```

BOOL (tipo Boolean)

Un valore booleano. Un valore booleano di DynamoDB è indicato da:

```
{ "BOOL" : true }
```

Solo i valori `true` e `false` sono validi.

Un esempio di utilizzo è:

```
"attributeValues" : {  
  "orderComplete" : { "BOOL" : false }  
}
```

L (tipo List)

Un elenco di qualsiasi altro valore DynamoDB supportato. Un valore di DynamoDB List è indicato da:

```
{ "L" : [ ... ] }
```


Nota che il valore è un valore composto, in cui l'elenco può contenere zero o più di qualsiasi valore DynamoDB supportato (inclusi altri elenchi). L'elenco può anche contenere una combinazione di diversi tipi.

Un esempio di utilizzo è:

```
{ "L" : [  
  { "S" : "A string value" },  
  { "N" : 1 },  
  { "SS" : [ "Another string value", "Even more string values!" ] }  
]
```

M (tipo Map)

Rappresenta una raccolta non ordinata di coppie chiave-valore di altri valori DynamoDB supportati. Un valore di DynamoDB Map è indicato da:

```
{ "M" : { ... } }
```

Una mappa può contenere zero o più coppie chiave-valore. La chiave deve essere una stringa e il valore può essere qualsiasi valore DynamoDB supportato (incluse altre mappe). La mappa può anche contenere una combinazione di diversi tipi.

Un esempio di utilizzo è:

```
{ "M" : {  
  "someString" : { "S" : "A string value" },  
  "someNumber" : { "N" : 1 },  
  "stringSet" : { "SS" : [ "Another string value", "Even more string  
values!" ] }  
}
```

NULL (tipo Null)

Un valore nullo. Un valore Null di DynamoDB è indicato da:

```
{ "NULL" : null }
```

Un esempio di utilizzo è:

```
"attributeValues" : {  
  "phoneNumbers" : { "NULL" : null }  
}
```

Per ulteriori informazioni su ciascun tipo, consulta la [documentazione di DynamoDB](#).

Sistema di tipi (mappatura delle risposte)

Quando riceve una risposta da DynamoDB AWS AppSync, la converte automaticamente in tipi primitivi GraphQL e JSON. Ogni attributo in DynamoDB viene decodificato e restituito nel contesto del gestore delle risposte.

Ad esempio, se DynamoDB restituisce quanto segue:

```
{  
  "id" : { "S" : "1234" },  
  "name" : { "S" : "Nadia" },  
  "age" : { "N" : 25 }  
}
```

Quando il risultato viene restituito dal tuo risolutore di pipeline, AWS AppSync lo converte in tipi GraphQL e JSON come:

```
{  
  "id" : "1234",  
  "name" : "Nadia",  
  "age" : 25  
}
```

Questa sezione spiega come AWS AppSync convertire i seguenti tipi di scalari, documenti e set di DynamoDB:

S (tipo String)

Un singolo valore di stringa. Un valore DynamoDB String viene restituito come stringa.

Ad esempio, se DynamoDB ha restituito il seguente valore DynamoDB String:

```
{ "S" : "some string" }
```

AWS AppSync lo converte in una stringa:

```
"some string"
```

SS (tipo String set)

Un set di valori di stringa. Un valore DynamoDB String Set viene restituito come elenco di stringhe.

Ad esempio, se DynamoDB ha restituito il seguente valore DynamoDB String Set:

```
{ "SS" : [ "first value", "second value", ... ] }
```

AWS AppSync lo converte in un elenco di stringhe:

```
[ "+1 555 123 4567", "+1 555 234 5678" ]
```

N (tipo Number)

Un singolo valore numerico. Un valore numerico di DynamoDB viene restituito come numero.

Ad esempio, se DynamoDB ha restituito il seguente valore numerico di DynamoDB:

```
{ "N" : 1234 }
```

AWS AppSync lo converte in un numero:

```
1234
```

NS (tipo Number set)

Un set di valori numerici. Un valore del set di numeri di DynamoDB viene restituito come elenco di numeri.

Ad esempio, se DynamoDB ha restituito il seguente valore del DynamoDB Number Set:

```
{ "NS" : [ 67.8, 12.2, 70 ] }
```

AWS AppSync lo converte in un elenco di numeri:

```
[ 67.8, 12.2, 70 ]
```

B (tipo Binary)

Un valore binario. Un valore binario di DynamoDB viene restituito come stringa contenente la rappresentazione in base64 di quel valore.

Ad esempio, se DynamoDB ha restituito il seguente valore binario di DynamoDB:

```
{ "B" : "SGVsbG8sIFdvcmxkIQo=" }
```

AWS AppSync lo converte in una stringa contenente la rappresentazione in base64 del valore:

```
"SGVsbG8sIFdvcmxkIQo="
```

I dati binari vengono codificati nello schema di codifica base64 secondo quanto specificato negli standard [RFC 4648](#) e [RFC 2045](#).

BS (tipo Binary set)

Un set di valori binari. Un valore del set binario di DynamoDB viene restituito come elenco di stringhe contenenti la rappresentazione in base64 dei valori.

Ad esempio, se DynamoDB ha restituito il seguente valore di DynamoDB Binary Set:

```
{ "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ] }
```

AWS AppSync lo converte in un elenco di stringhe contenenti la rappresentazione in base64 dei valori:

```
[ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ]
```

I dati binari vengono codificati nello schema di codifica base64 secondo quanto specificato negli standard [RFC 4648](#) e [RFC 2045](#).

BOOL (tipo Boolean)

Un valore booleano. Un valore booleano DynamoDB viene restituito come booleano.

Ad esempio, se DynamoDB ha restituito il seguente valore booleano di DynamoDB:

```
{ "BOOL" : true }
```

AWS AppSync lo converte in booleano:

```
true
```

L (tipo List)

Un elenco di qualsiasi altro valore DynamoDB supportato. Un valore DynamoDB List viene restituito come elenco di valori, in cui viene convertito anche ogni valore interno.

Ad esempio, se DynamoDB ha restituito il seguente valore di DynamoDB List:

```
{ "L" : [
  { "S" : "A string value" },
  { "N" : 1 },
  { "SS" : [ "Another string value", "Even more string values!" ] }
]
```

AWS AppSync lo converte in un elenco di valori convertiti:

```
[ "A string value", 1, [ "Another string value", "Even more string values!" ] ]
```

M (tipo Map)

Una key/value raccolta di qualsiasi altro valore DynamoDB supportato. Un valore della mappa DynamoDB viene restituito come oggetto JSON, dove key/value ogni valore viene anche convertito.

Ad esempio, se DynamoDB ha restituito il seguente valore di DynamoDB Map:

```
{ "M" : {
  "someString" : { "S" : "A string value" },
  "someNumber" : { "N" : 1 },
  "stringSet" : { "SS" : [ "Another string value", "Even more string
values!" ] }
}
```

AWS AppSync lo converte in un oggetto JSON:

```
{
  "someString" : "A string value",
  "someNumber" : 1,
  "stringSet" : [ "Another string value", "Even more string values!" ]
}
```

NULL (tipo Null)

Un valore nullo.

Ad esempio, se DynamoDB ha restituito il seguente valore Null di DynamoDB:

```
{ "NULL" : null }
```

AWS AppSync lo converte in un valore nullo:

```
null
```

Filtri

Quando si eseguono interrogazioni su oggetti in DynamoDB utilizzando Query le operazioni Scan and, è possibile facoltativamente specificare `filter` un valore che valuti i risultati e restituisca solo i valori desiderati.

La proprietà `filter` di una Scan richiesta Query or ha la seguente struttura:

```
type DynamoDBExpression = {
  expression: string;
  expressionNames?: { [key: string]: string };
  expressionValues?: { [key: string]: any };
};
```

I campi sono definiti come segue:

expression

L'espressione della query. Per ulteriori informazioni su come scrivere espressioni di filtro, consulta la documentazione di [DynamoDB e QueryFilter ScanFilter](#) DynamoDB. Questo campo deve essere specificato.

expressionNames

Le sostituzioni per i segnaposto dell'attributo di espressione name sotto forma di coppie chiave-valore. La chiave corrisponde a un segnaposto nome utilizzato in `expression`. Il valore deve essere una stringa che corrisponde al nome dell'attributo dell'elemento in DynamoDB. Questo è un campo facoltativo in cui vanno riportate solo le sostituzioni per i segnaposto dell'attributo di espressione name utilizzate in `expression`.

expressionValues

Le sostituzioni per i segnaposto dell'attributo di espressione value sotto forma di coppie chiave-valore. La chiave corrisponde a un segnaposto per un valore utilizzato in `expression`, mentre il valore deve essere un valore tipizzato. Per ulteriori informazioni su come specificare un «valore digitato», consulta [Type system \(request mapping\)](#). Questo elemento deve essere specificato. Questo è un campo facoltativo in cui vanno riportate solo le sostituzioni per i segnaposto dell'attributo di espressione value utilizzate in `expression`.

Esempio

L'esempio seguente è una sezione di filtro per una richiesta, in cui le voci recuperate da DynamoDB vengono restituite solo se il titolo inizia con l'argomento. `title`

Qui utilizziamo il `util.transform.toDynamoDBFilterExpression` per creare automaticamente un filtro da un oggetto:

```
const filter = util.transform.toDynamoDBFilterExpression({
  title: { beginsWith: 'far away' },
});

const request = {};
request.filter = JSON.parse(filter);
```

Questo genera il seguente filtro:

```
{
  "filter": {
    "expression": "(begins_with(#title,:title_beginsWith))",
    "expressionNames": { "#title": "title" },
    "expressionValues": {
      ":title_beginsWith": { "S": "far away" }
```

```
    }  
  }  
}
```

Espressioni di condizione

Quando modificate gli oggetti in DynamoDB utilizzando le operazioni `PutItem`, e `DeleteItem` `DynamoDBUpdateItem`, potete facoltativamente specificare un'espressione di condizione che controlli se la richiesta deve avere successo o meno, in base allo stato dell'oggetto già in DynamoDB prima dell'esecuzione dell'operazione.

La funzione AWS AppSync DynamoDB consente di specificare un'espressione di condizione `PutItem` negli `UpdateItem` oggetti `DeleteItem` e di richiedere una strategia da seguire se la condizione fallisce e l'oggetto non è stato aggiornato.

Esempio 1

Il seguente oggetto `PutItem` di richiesta non ha un'espressione di condizione. Di conseguenza, inserisce un elemento in DynamoDB anche se esiste già un elemento con la stessa chiave, sovrascrivendo così l'elemento esistente.

```
import { util } from '@aws-appsync/utils';  
export function request(ctx) {  
  const { foo, bar, ...values } = ctx.args  
  return {  
    operation: 'PutItem',  
    key: util.dynamodb.toMapValues({foo, bar}),  
    attributeValues: util.dynamodb.toMapValues(values),  
  };  
}
```

Esempio 2

L'`PutItem` oggetto seguente ha un'espressione di condizione che consente la riuscita dell'operazione solo se un elemento con la stessa chiave non esiste in DynamoDB.

```
import { util } from '@aws-appsync/utils';  
export function request(ctx) {  
  const { foo, bar, ...values } = ctx.args  
  return {  
    operation: 'PutItem',  
    conditionExpression: 'attribute_not_exists(key)',  
    key: util.dynamodb.toMapValues({foo, bar}),  
    attributeValues: util.dynamodb.toMapValues(values),  
  };  
}
```



```
key: util.dynamodb.toMapValues({foo, bar}),
attributeValues: util.dynamodb.toMapValues(values),
condition: { expression: "attribute_not_exists(id)" }
};
}
```

Per impostazione predefinita, se il controllo delle condizioni fallisce, la funzione AWS AppSync DynamoDB fornisce un errore per la mutazione.

Tuttavia, la funzione AWS AppSync DynamoDB offre alcune funzionalità aggiuntive per aiutare gli sviluppatori a gestire alcuni casi limite comuni:

- Se AWS AppSync le funzioni di DynamoDB sono in grado di determinare che il valore corrente in DynamoDB corrisponde al risultato desiderato, considerano l'operazione come se fosse riuscita comunque.
- Invece di restituire un errore, è possibile configurare la funzione per richiamare una funzione Lambda personalizzata per decidere come la funzione AWS AppSync DynamoDB deve gestire l'errore.

Questi sono descritti più dettagliatamente nella sezione [Gestione di un errore di controllo delle condizioni](#).

[Per ulteriori informazioni sulle espressioni delle condizioni di DynamoDB, consulta la documentazione di DynamoDB. ConditionExpressions](#)

Specificare una condizione

Gli oggetti `PutItemUpdateItem`, e `DeleteItem` request consentono tutti di specificare una `condition` sezione facoltativa. Se omessa, non vengono eseguiti controlli di condizione. Se specificata, la condizione deve essere soddisfatta perché l'operazione abbia esito positivo.

Una sezione `condition` ha la seguente struttura:

```
type ConditionCheckExpression = {
  expression: string;
  expressionNames?: { [key: string]: string };
  expressionValues?: { [key: string]: any };
  equalsIgnore?: string[];
  consistentRead?: boolean;
  conditionalCheckFailedHandler?: {
```

```
strategy: 'Custom' | 'Reject';
lambdaArn?: string;
};
};
```

I seguenti campi specificano la condizione:

expression

L'espressione di aggiornamento in sé. Per ulteriori informazioni su come scrivere espressioni di condizione, consulta la documentazione di [ConditionExpressions DynamoDB](#). Questo campo deve essere specificato.

expressionNames

Le sostituzioni per i segnaposto dell'attributo di espressione name, sotto forma di coppie chiave-valore. La chiave corrisponde a un segnaposto per il nome utilizzato nell'espressione e il valore deve essere una stringa corrispondente al nome dell'attributo dell'elemento in DynamoDB. Questo è un campo facoltativo in cui vanno riportate solo le sostituzioni per i segnaposto dell'attributo di espressione name utilizzate in expression.

expressionValues

Le sostituzioni per i segnaposto del valore dell'attributo di espressione, sotto forma di coppie chiave-valore. La chiave corrisponde a un segnaposto per un valore utilizzato nell'espressione, mentre il valore deve essere un valore tipizzato. Per ulteriori informazioni su come specificare un «valore digitato», consulta [Type system](#) (request mapping). Questo elemento deve essere specificato. Questo è un campo facoltativo in cui vanno riportate solo le sostituzioni per i segnaposto dell'attributo di espressione value utilizzate nell'espressione.

I campi rimanenti indicano alla funzione AWS AppSync DynamoDB come gestire un errore di controllo delle condizioni:

equalsIgnore

Quando un controllo delle condizioni fallisce durante l'utilizzo dell'PutItem operazione, la funzione AWS AppSync DynamoDB confronta l'elemento attualmente in DynamoDB con l'elemento che ha cercato di scrivere. Se sono uguali, tratta l'operazione come se avesse avuto comunque esito positivo. È possibile utilizzare il equalsIgnore campo per specificare un elenco di attributi da AWS AppSync ignorare durante l'esecuzione del confronto. Ad esempio, se l'unica differenza era un version attributo, considera l'operazione come se fosse riuscita. Questo campo è facoltativo.

consistentRead

Quando un controllo delle condizioni fallisce, AWS AppSync ottiene il valore corrente dell'elemento da DynamoDB utilizzando una lettura fortemente coerente. È possibile utilizzare questo campo per indicare alla funzione AWS AppSync DynamoDB di utilizzare invece una lettura alla fine coerente. Si tratta di un campo facoltativo, impostato di default su `true`.

conditionalCheckFailedHandler

Questa sezione consente di specificare in che modo la funzione AWS AppSync DynamoDB tratta un errore di controllo delle condizioni dopo aver confrontato il valore corrente in DynamoDB con il risultato previsto. Questa sezione è facoltativa. Se omissa, la strategia predefinita è `Reject`.

strategy

La strategia adottata dalla AWS AppSync funzione DynamoDB dopo aver confrontato il valore corrente in DynamoDB con il risultato previsto. Questo campo è obbligatorio e ha i seguenti possibili valori:

Reject

La mutazione fallisce e viene aggiunto un errore alla risposta GraphQL.

Custom

La funzione AWS AppSync DynamoDB richiama una funzione Lambda personalizzata per decidere come gestire l'errore del controllo delle condizioni. Se `strategy` è impostato su `Custom`, il campo `lambdaArn` deve contenere l'ARN della funzione Lambda da invocare.

lambdaArn

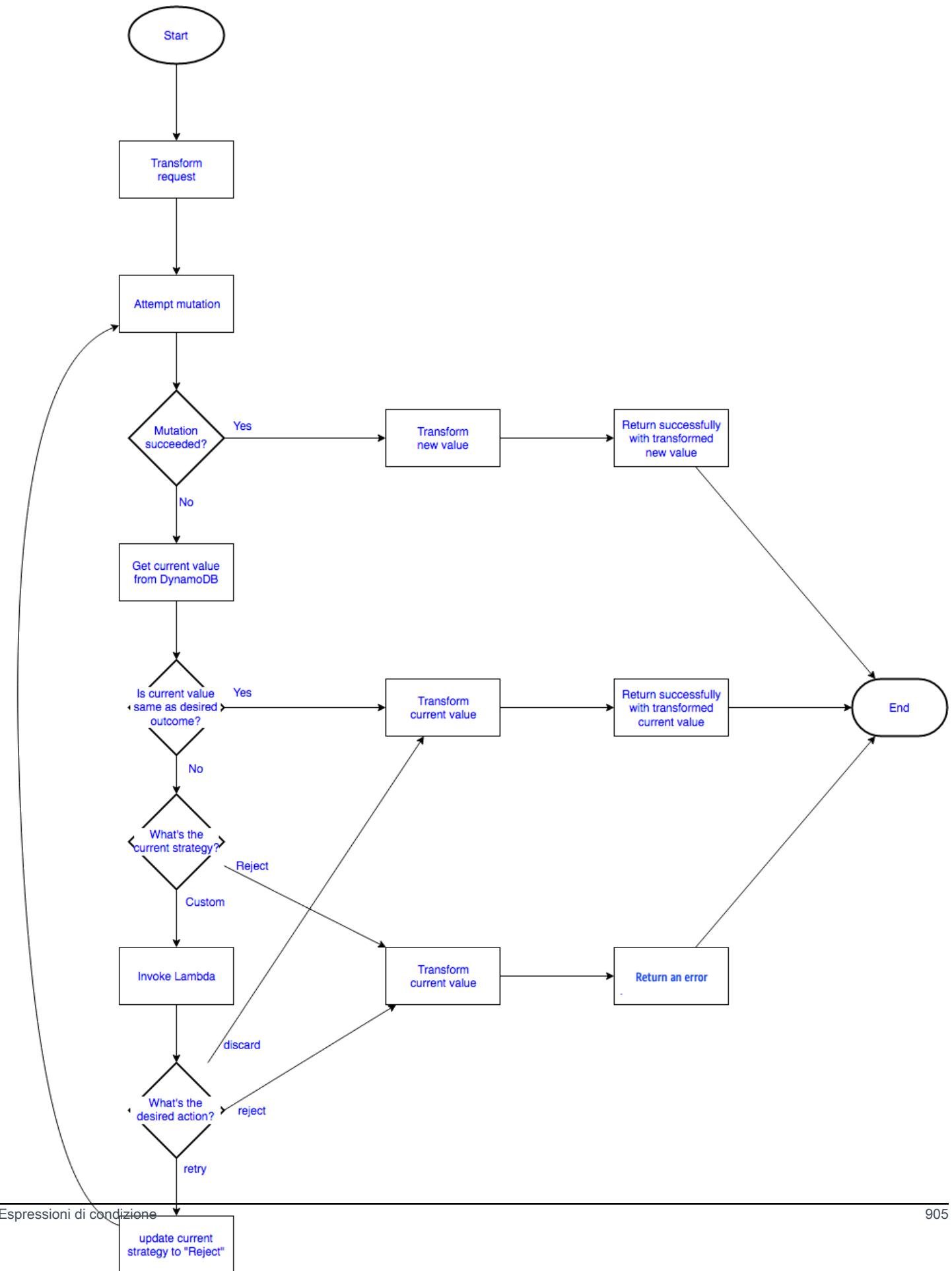
L'ARN della funzione Lambda da richiamare che determina come la funzione DynamoDB deve gestire l'errore del controllo delle AWS AppSync condizioni. Questo campo deve essere specificato solo se `strategy` è impostato su `Custom`. Per ulteriori informazioni su come utilizzare questa funzionalità, consulta [Gestione](#) di un errore di controllo delle condizioni.

Gestione di un errore di controllo delle condizioni

Quando un controllo delle condizioni fallisce, la AWS AppSync funzione DynamoDB può trasmettere l'errore per la mutazione e il valore corrente dell'oggetto utilizzando l'utilità `util.appendError`. Tuttavia, la funzione AWS AppSync DynamoDB offre alcune funzionalità aggiuntive per aiutare gli sviluppatori a gestire alcuni casi limite comuni:

- Se AWS AppSync le funzioni di DynamoDB sono in grado di determinare che il valore corrente in DynamoDB corrisponde al risultato desiderato, considerano l'operazione come se fosse riuscita comunque.
- Invece di restituire un errore, è possibile configurare la funzione per richiamare una funzione Lambda personalizzata per decidere come la funzione AWS AppSync DynamoDB deve gestire l'errore.

Il diagramma di questo processo è il seguente:



Verifica del risultato desiderato

Quando il controllo delle condizioni fallisce, la funzione AWS AppSync DynamoDB esegue una richiesta `GetItem` DynamoDB per ottenere il valore corrente dell'elemento da DynamoDB. Per impostazione predefinita, si utilizza una lettura fortemente consistente, ma è possibile intervenire sulla configurazione utilizzando il campo `consistentRead` del blocco `condition` e confrontandolo con il risultato previsto:

- Per l'operazione `PutItem`, la funzione AWS AppSync DynamoDB confronta il valore corrente con quello che ha tentato di scrivere, escludendo dal confronto tutti gli attributi elencati. `equalsIgnore` Se gli elementi sono uguali, considera l'operazione come riuscita e restituisce l'elemento recuperato da DynamoDB. In caso contrario, viene seguita la strategia configurata.

Ad esempio, se l'oggetto della `PutItem` richiesta aveva il seguente aspetto:

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { id, name, version } = ctx.args
  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({foo, bar}),
    attributeValues: util.dynamodb.toMapValues({ name, version: version+1 }),
    condition: {
      expression: "version = :expectedVersion",
      expressionValues: util.dynamodb.toMapValues({':expectedVersion': version}),
      equalsIgnore: ['version']
    }
  };
}
```

E la voce attualmente inclusa in DynamoDB fosse simile alla seguente:

```
{
  "id" : { "S" : "1" },
  "name" : { "S" : "Steve" },
  "version" : { "N" : 8 }
}
```

La funzione AWS AppSync DynamoDB confronta l'elemento che ha provato a scrivere con il valore corrente, verifica che l'unica differenza è `version` il campo, ma poiché è configurata per ignorare

version il campo, considera l'operazione come riuscita e restituisce l'elemento recuperato da DynamoDB.

- Per l'`DeleteItem` operazione, la funzione AWS AppSync DynamoDB verifica che un elemento sia stato restituito da DynamoDB. Se nessuna voce è stata restituita, considera l'operazione riuscita. In caso contrario, viene seguita la strategia configurata.
- Per l'`UpdateItem` operazione, la funzione AWS AppSync DynamoDB non dispone di informazioni sufficienti per determinare se l'elemento attualmente in DynamoDB corrisponde al risultato previsto e pertanto segue la strategia configurata.

Se lo stato corrente dell'oggetto in DynamoDB è diverso dal risultato previsto, la funzione AWS AppSync DynamoDB segue la strategia configurata, rifiutando la mutazione o richiamando una funzione Lambda per determinare cosa fare dopo.

Seguendo la strategia di «rifiuto»

Quando si segue la `Reject` strategia, la funzione AWS AppSync DynamoDB restituisce un errore per la mutazione.

Ad esempio, partendo dalla richiesta di mutazione seguente:

```
mutation {  
  updatePerson(id: 1, name: "Steve", expectedVersion: 1) {  
    Name  
    theVersion  
  }  
}
```

Se l'elemento restituito da DynamoDB è simile al seguente:

```
{  
  "id" : { "S" : "1" },  
  "name" : { "S" : "Steve" },  
  "version" : { "N" : 8 }  
}
```

E il gestore della risposta alla funzione ha il seguente aspetto:

```
import { util } from '@aws-appsync/utils';
```

```
export function response(ctx) {
  const { version, ...values } = ctx.result;
  const result = { ...values, theVersion: version };
  if (ctx.error) {
    if (error) {
      return util.appendError(error.message, error.type, result, null);
    }
  }
  return result
}
```

La risposta GraphQL ha il seguente aspetto:

```
{
  "data": null,
  "errors": [
    {
      "message": "The conditional request failed (Service: AmazonDynamoDBv2;
Status Code: 400; Error Code: ConditionalCheckFailedException; Request ID:
ABCDEFGHJKLMNOPQRSTUVWXYZABCDEFGHJKLMNOPQRSTUVWXYZ)"
      "errorType": "DynamoDB:ConditionalCheckFailedException",
      ...
    }
  ]
}
```

Inoltre, se i campi dell'oggetto restituito sono compilati da altri resolver in caso di successo della mutazione, tali campi non saranno risolti quando l'oggetto viene restituito nella sezione `error`.

Seguendo la strategia «personalizzata»

Quando si segue la Custom strategia, la funzione AWS AppSync DynamoDB richiama una funzione Lambda per decidere cosa fare dopo. La funzione Lambda sceglie una delle seguenti opzioni:

- Rifiuto della mutazione (`reject`). Ciò indica alla funzione AWS AppSync DynamoDB di comportarsi come se lo `Reject` fosse la strategia configurata, restituendo un errore per la mutazione e il valore corrente dell'oggetto in DynamoDB come descritto nella sezione precedente.
- Rifiuto della mutazione (`discard`). Ciò indica alla funzione AWS AppSync DynamoDB di ignorare silenziosamente l'errore del controllo delle condizioni e restituisce il valore in DynamoDB.
- Rifiuto della mutazione (`retry`). Questo indica alla funzione AWS AppSync DynamoDB di ritentare la mutazione con un nuovo oggetto di richiesta.

La richiesta di invocazione Lambda

La funzione AWS AppSync DynamoDB richiama la funzione Lambda specificata in. `lambdaArn`. Utilizza lo stesso `service-role-arn` configurato per l'origine dati. Il payload dell'invocazione ha la seguente struttura:

```
{
  "arguments": { ... },
  "requestMapping": {... },
  "currentValue": { ... },
  "resolver": { ... },
  "identity": { ... }
}
```

I campi sono definiti come segue:

arguments

Gli argomenti della mutazione GraphQL. È lo stesso degli argomenti disponibili per l'oggetto della richiesta in. `context.arguments`

requestMapping

L'oggetto di richiesta per questa operazione.

currentValue

Il valore corrente dell'oggetto in DynamoDB.

resolver

Informazioni sul AWS AppSync resolver o sulla funzione.

identity

Informazioni sul chiamante. Sono le stesse informazioni sull'identità disponibili per l'oggetto della richiesta in. `context.identity`

Un esempio completo di payload:

```
{
  "arguments": {
```

```

    "id": "1",
    "name": "Steve",
    "expectedVersion": 1
  },
  "requestMapping": {
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
      "id" : { "S" : "1" }
    },
    "attributeValues" : {
      "name" : { "S" : "Steve" },
      "version" : { "N" : 2 }
    },
    "condition" : {
      "expression" : "version = :expectedVersion",
      "expressionValues" : {
        ":expectedVersion" : { "N" : 1 }
      },
      "equalsIgnore": [ "version" ]
    }
  },
  "currentValue": {
    "id" : { "S" : "1" },
    "name" : { "S" : "Steve" },
    "version" : { "N" : 8 }
  },
  "resolver": {
    "tableName": "People",
    "awsRegion": "us-west-2",
    "parentType": "Mutation",
    "field": "updatePerson",
    "outputType": "Person"
  },
  "identity": {
    "accountId": "123456789012",
    "sourceIp": "x.x.x.x",
    "user": "AIDAAAAAAAAAAAAAAAAAAAA",
    "userArn": "arn:aws:iam::123456789012:user/appsync"
  }
}

```

La risposta all'invocazione Lambda

La funzione Lambda può ispezionare il payload di chiamata e applicare qualsiasi logica aziendale per decidere come la funzione DynamoDB deve gestire l'errore AWS AppSync . Sono disponibili tre opzioni per la gestione dell'errore nel controllo della condizione:

- Rifiuto della mutazione (`reject`). Il payload di risposta per questa opzione deve avere questa struttura:

```
{
  "action": "reject"
}
```

Ciò indica alla funzione AWS AppSync DynamoDB di comportarsi come se la strategia configurata lo `Reject` fosse, restituendo un errore per la mutazione e il valore corrente dell'oggetto in DynamoDB, come descritto nella sezione precedente.

- Rifiuto della mutazione (`discard`). Il payload di risposta per questa opzione deve avere questa struttura:

```
{
  "action": "discard"
}
```

Ciò indica alla funzione AWS AppSync DynamoDB di ignorare silenziosamente l'errore del controllo delle condizioni e restituisce il valore in DynamoDB.

- Rifiuto della mutazione (`retry`). Il payload di risposta per questa opzione deve avere questa struttura:

```
{
  "action": "retry",
  "retryMapping": { ... }
}
```

Questo indica alla funzione AWS AppSync DynamoDB di ritentare la mutazione con un nuovo oggetto di richiesta. La struttura della `retryMapping` sezione dipende dall'operazione DynamoDB ed è un sottoinsieme dell'oggetto di richiesta completo per tale operazione.

Per `PutItem`, la sezione `retryMapping` ha la seguente struttura. Per una descrizione del `attributeValues` campo, vedere. [PutItem](#)

```
{
  "attributeValues": { ... },
  "condition": {
    "equalsIgnore" = [ ... ],
    "consistentRead" = true
  }
}
```

Per `UpdateItem`, la sezione `retryMapping` ha la seguente struttura. Per una descrizione della update sezione, vedere [UpdateItem](#).

```
{
  "update" : {
    "expression" : "someExpression"
    "expressionNames" : {
      "#foo" : "foo"
    },
    "expressionValues" : {
      ":bar" : ... typed value
    }
  },
  "condition": {
    "consistentRead" = true
  }
}
```

Per `DeleteItem`, la sezione `retryMapping` ha la seguente struttura.

```
{
  "condition": {
    "consistentRead" = true
  }
}
```

Non c'è modo di specificare operazioni o chiavi diverse su cui lavorare. La funzione AWS AppSync DynamoDB consente solo nuovi tentativi della stessa operazione sullo stesso oggetto. Inoltre, la sezione `condition` non consente di specificare un valore per `conditionalCheckFailedHandler`. Se il nuovo tentativo fallisce, la funzione AWS AppSync DynamoDB segue la strategia. `Reject`

Ecco un esempio di funzione Lambda per far fronte a una richiesta PutItem non riuscita. La logica di business osserva l'autore della chiamata. Se è stata effettuata da `jeffTheAdmin`, riprova la richiesta, aggiornando l'`version` e `expectedVersion` dall'elemento attualmente in DynamoDB. Altrimenti rifiuta la mutazione.

```
exports.handler = (event, context, callback) => {
  console.log("Event: " + JSON.stringify(event));

  // Business logic goes here.

  var response;
  if ( event.identity.user == "jeffTheAdmin" ) {
    response = {
      "action" : "retry",
      "retryMapping" : {
        "attributeValues" : event.requestMapping.attributeValues,
        "condition" : {
          "expression" : event.requestMapping.condition.expression,
          "expressionValues" :
            event.requestMapping.condition.expressionValues
        }
      }
    }
    response.retryMapping.attributeValues.version = { "N" :
      event.currentValue.version.N + 1 }
    response.retryMapping.condition.expressionValues[':expectedVersion'] =
      event.currentValue.version
  } else {
    response = { "action" : "reject" }
  }

  console.log("Response: " + JSON.stringify(response))
  callback(null, response)
};
```

Espressioni relative alle condizioni delle transazioni

Le espressioni delle condizioni di transazione sono disponibili nelle richieste di tutti e quattro i tipi di operazioni in `TransactWriteItems`, vale a dire `PutItem`, `DeleteItem`, `UpdateItem`, e `ConditionCheck`.

PerPutItem, e DeleteItemUpdateItem, l'espressione della condizione di transazione è facoltativa. PerchéConditionCheck, l'espressione della condizione della transazione è obbligatoria.

Esempio 1

Il seguente gestore di richieste di DeleteItem funzioni transazionali non dispone di un'espressione di condizione. Di conseguenza, elimina l'elemento in DynamoDB.

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { postId } = ctx.args;
  return {
    operation: 'TransactWriteItems',
    transactItems: [
      {
        table: 'posts',
        operation: 'DeleteItem',
        key: util.dynamodb.toMapValues({ postId }),
      }
    ],
  };
}
```

Esempio 2

Il seguente gestore di richieste di DeleteItem funzioni transazionali dispone di un'espressione della condizione di transazione che consente l'esito positivo dell'operazione solo se l'autore di quel post è uguale a un determinato nome.

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { postId, authorName } = ctx.args;
  return {
    operation: 'TransactWriteItems',
    transactItems: [
      {
        table: 'posts',
        operation: 'DeleteItem',
        key: util.dynamodb.toMapValues({ postId }),
        condition: util.transform.toDynamoDBConditionExpression({
```

```

        authorName: { eq: authorName },
      })),
    ],
  };
}
```

Se il controllo della condizione fallisce, genera una `TransactionCanceledException` e il dettaglio dell'errore viene restituito in `ctx.result.cancellationReasons`. Tieni presente che, per impostazione predefinita, verrà restituito il vecchio elemento in DynamoDB che ha impedito il controllo delle condizioni. `ctx.result.cancellationReasons`

Specificare una condizione

Gli oggetti `PutItemUpdateItem`, e `DeleteItem` request consentono tutti di specificare una `condition` sezione facoltativa. Se omessa, non vengono eseguiti controlli di condizione. Se specificata, la condizione deve essere soddisfatta perché l'operazione abbia esito positivo. Per `ConditionCheck` deve essere specificata una sezione `condition`. La condizione deve essere vera affinché l'intera transazione abbia esito positivo.

Una sezione `condition` ha la seguente struttura:

```

type TransactConditionCheckExpression = {
  expression: string;
  expressionNames?: { [key: string]: string };
  expressionValues?: { [key: string]: string };
  returnValuesOnConditionCheckFailure: boolean;
};
```

I seguenti campi specificano la condizione:

expression

L'espressione di aggiornamento in sé. Per ulteriori informazioni su come scrivere espressioni di condizione, consulta la documentazione di [ConditionExpressions DynamoDB](#). Questo campo deve essere specificato.

expressionNames

Le sostituzioni per i segnaposto dell'attributo di espressione name, sotto forma di coppie chiave-valore. La chiave corrisponde a un segnaposto per il nome utilizzato nell'espressione e il valore

deve essere una stringa corrispondente al nome dell'attributo dell'elemento in DynamoDB. Questo è un campo facoltativo in cui vanno riportate solo le sostituzioni per i segnaposto dell'attributo di espressione name utilizzate inexpression.

expressionValues

Le sostituzioni per i segnaposto del valore dell'attributo di espressione, sotto forma di coppie chiave-valore. La chiave corrisponde a un segnaposto per un valore utilizzato nell'espressione, mentre il valore deve essere un valore tipizzato. Per ulteriori informazioni su come specificare un «valore digitato», consulta [Type system](#) (request mapping). Questo elemento deve essere specificato. Questo è un campo facoltativo in cui vanno riportate solo le sostituzioni per i segnaposto dell'attributo di espressione value utilizzate nell'espressione.

returnValuesOnConditionCheckFailure

Specificare se recuperare l'elemento in DynamoDB quando un controllo delle condizioni fallisce. L'elemento recuperato sarà in `ctx.result.cancellationReasons[<index>].item`, dove `<index>` è l'indice dell'elemento richiesta che non ha superato il controllo della condizione. Il valore predefinito di questo valore è `true`.

Proiezioni

Quando si leggono oggetti in DynamoDB utilizzando `GetItem` le operazioni `Scan`, `QueryBatchGetItem`, `TransactGetItems` e, è possibile specificare facoltativamente una proiezione che identifichi gli attributi desiderati. La proprietà di proiezione ha la seguente struttura, che è simile ai filtri:

```
type DynamoDBExpression = {  
  expression: string;  
  expressionNames?: { [key: string]: string }  
};
```

I campi sono definiti come segue:

expression

L'espressione di proiezione, che è una stringa. Per recuperare un singolo attributo, specificane il nome. Per più attributi, i nomi devono essere valori separati da virgole. Per ulteriori informazioni sulla scrittura di espressioni di proiezione, consulta la documentazione delle espressioni di proiezione [DynamoDB](#). Questo campo è obbligatorio.

expressionNames

Le sostituzioni degli attributi di espressione chiamano segnaposto sotto forma di coppie chiave-valore. La chiave corrisponde a un segnaposto nome utilizzato in `expression`. Il valore deve essere una stringa che corrisponde al nome dell'attributo dell'elemento in DynamoDB. Questo campo è facoltativo e deve essere compilato solo con sostituzioni per i segnaposto dei nomi degli attributi di espressione utilizzati in `expression`. Per ulteriori informazioni su `expressionNames`, consulta la documentazione di [DynamoDB](#).

Esempio 1

L'esempio seguente è una sezione di proiezione per una JavaScript funzione in cui solo gli attributi `author` e `id` vengono restituiti da DynamoDB:

```
projection : {  
  expression : "#author, id",  
  expressionNames : {  
    "#author" : "author"  
  }  
}
```

Tip

È possibile accedere al set di selezione delle richieste GraphQL utilizzando [selectionSetList](#). Questo campo consente di inquadrare l'espressione di proiezione in modo dinamico in base alle proprie esigenze.

Note

Quando si utilizzano espressioni di proiezione con le Scan operazioni Query and, il valore per `select` deve essere `SPECIFIC_ATTRIBUTES`. Per ulteriori informazioni, consulta la documentazione di [DynamoDB](#).

AWS AppSync JavaScript riferimento alla funzione resolver per OpenSearch

Il AWS AppSync resolver per Amazon OpenSearch Service ti consente di utilizzare GraphQL per archiviare e recuperare dati nei domini di OpenSearch servizio esistenti nel tuo account. Questo resolver consente di mappare una richiesta GraphQL in OpenSearch entrata in una richiesta di servizio e quindi mappare la risposta OpenSearch del servizio a GraphQL. Questa sezione descrive i gestori di richieste e risposte delle funzioni per le operazioni di servizio supportate. OpenSearch

Richiesta

La maggior parte degli oggetti OpenSearch Service Request ha una struttura comune in cui cambiano solo pochi pezzi. L'esempio seguente esegue una ricerca in base a un dominio OpenSearch Service, in cui i documenti sono di tipo post e sotto indicizzati. id I parametri di ricerca sono definiti nella sezione body, con molte delle clausole di query comuni definite nel campo query. Questo esempio esegue la ricerca di documenti contenenti "Nadia" o "Bailey", o entrambe le stringhe, nel campo author di un documento:

```
export function request(ctx) {
  return {
    operation: 'GET',
    path: '/id/post/_search',
    params: {
      headers: {},
      queryString: {},
      body: {
        from: 0,
        size: 50,
        query: {
          bool: {
            should: [
              { match: { author: 'Nadia' } },
              { match: { author: 'Bailey' } },
            ],
          },
        },
      },
    },
  };
}
```

Risposta

Come con altre fonti di dati, OpenSearch Service invia una risposta AWS AppSync che deve essere convertita in GraphQL.

La maggior parte delle query GraphQL cerca il `_source` campo in una OpenSearch risposta del servizio. Poiché è possibile eseguire ricerche per restituire un singolo documento o un elenco di documenti, in Service vengono utilizzati due modelli di risposta comuni: OpenSearch

Elenco di risultati

```
export function response(ctx) {  
  const entries = [];  
  for (const entry of ctx.result.hits.hits) {  
    entries.push(entry['_source']);  
  }  
  return entries;  
}
```

Singola voce

```
export function response(ctx) {  
  return ctx.result['_source']  
}
```

Campo **operation**

Note

Questo vale solo per il gestore delle richieste.

Metodo o verbo HTTP (GET, POST, PUT, HEAD o DELETE) che AWS AppSync invia al dominio del OpenSearch servizio. Sia la chiave che il valore devono essere stringhe.

```
"operation" : "PUT"
```

Campo **path**

Note

Questo vale solo per il gestore delle richieste.

Il percorso di ricerca per una richiesta OpenSearch di servizio da AWS AppSync. Costituisce un URL per il verbo HTTP dell'operazione. Sia la chiave che il valore devono essere stringhe.

```
"path" : "/indexname/type"
```

```
"path" : "/indexname/type/_search"
```

Quando il gestore della richiesta viene valutato, questo percorso viene inviato come parte della richiesta HTTP, incluso il dominio del OpenSearch servizio. L'esempio precedente potrebbe diventare:

```
GET https://opensearch-domain-name.REGION.es.amazonaws.com/indexname/type/_search
```

Campo **params**

Note

Questo vale solo per il gestore delle richieste.

Usato per specificare l'operazione eseguita dalla ricerca, in genere impostando il valore query all'interno di body. Ci sono tuttavia numerose altre funzionalità che è possibile configurare, ad esempio la formattazione delle risposte.

- headers

Informazioni dell'intestazione, come coppie chiave-valore. Sia la chiave che il valore devono essere stringhe. Esempio:

```
"headers" : {  
  "Content-Type" : "application/json"  
}
```

 Note

AWS AppSync attualmente supporta solo JSON come Content-Type

- queryString

Coppie chiave-valore che specificano opzioni comuni, ad esempio la formattazione del codice per le risposte JSON. Sia la chiave che il valore devono essere stringhe. Ad esempio, per ottenere codice JSON con formattazione Pretty, usa:

```
"queryString" : {  
  "pretty" : "true"  
}
```

- body

Questa è la parte principale della richiesta, che consente di AWS AppSync creare una richiesta di ricerca ben formata per il dominio del OpenSearch Servizio. La chiave deve essere una stringa costituita da un oggetto. Di seguito sono illustrati un paio di esempi.

Esempio 1

Restituisce tutti i documenti in cui la città corrisponde a "seattle":

```
export function request(ctx) {  
  return {  
    operation: 'GET',  
    path: '/id/post/_search',  
    params: {  
      headers: {},  
      queryString: {},  
      body: { from: 0, size: 50, query: { match: { city: 'seattle' } } },  
    },  
  };  
}
```

Esempio 2

Restituisce tutti i documenti in cui la città o lo stato corrisponde a "washington":

```
export function request(ctx) {
  return {
    operation: 'GET',
    path: '/id/post/_search',
    params: {
      headers: {},
      queryString: {},
      body: {
        from: 0,
        size: 50,
        query: {
          multi_match: { query: 'washington', fields: ['city', 'state'] },
        },
      },
    },
  };
}
```

Passare variabili

Note

Questo vale solo per il gestore di richieste.

Puoi anche passare variabili come parte della valutazione nel tuo gestore delle richieste. Supponi ad esempio di avere una query GraphQL come la seguente:

```
query {
  searchForState(state: "washington"){
    ...
  }
}
```

Il gestore della richiesta di funzione potrebbe essere il seguente:

```
export function request(ctx) {
  return {
    operation: 'GET',
    path: '/id/post/_search',
    params: {
```

```
headers: {},
queryString: {},
body: {
  from: 0,
  size: 50,
  query: {
    multi_match: { query: ctx.args.state, fields: ['city', 'state'] },
  },
},
};
}
```

AWS AppSync JavaScript riferimento alla funzione resolver per Lambda

Puoi usare AWS AppSync funzioni e resolver per richiamare le funzioni Lambda presenti nel tuo account. Puoi modellare i payload della richiesta e la risposta delle tue funzioni Lambda prima di restituirli ai tuoi clienti. Puoi anche specificare il tipo di operazione da eseguire nell'oggetto della richiesta. Questa sezione descrive le richieste per le operazioni Lambda supportate.

Oggetto Request

L'oggetto di richiesta Lambda gestisce i campi relativi alla tua funzione Lambda:

```
export type LambdaRequest = {
  operation: 'Invoke' | 'BatchInvoke';
  invocationType?: 'RequestResponse' | 'Event';
  payload: unknown;
};
```

Ecco un esempio che utilizza un'invokeoperazione i cui dati di payload sono il `getPost` campo di uno schema GraphQL insieme ai suoi argomenti dal contesto:

```
export function request(ctx) {
  return {
    operation: 'Invoke',
    payload: { field: 'getPost', arguments: ctx.args },
  };
}
```

```
}
```

L'intero documento di mappatura viene passato come input alla funzione Lambda in modo che l'esempio precedente abbia ora il seguente aspetto:

```
{
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "arguments": {
      "input": {
        "id": "postId1",
      }
    }
  }
}
```

Operation

L'origine dati Lambda consente di definire due operazioni sul `operation` campo: `Invoke` e `BatchInvoke`. L'`Invoke` operazione consente di AWS AppSync sapere di chiamare la funzione Lambda per ogni resolver di campi GraphQL. `BatchInvoke` indica di eseguire AWS AppSync in batch le richieste per il campo GraphQL corrente. Il campo `operation` è obbligatorio.

Infatti `Invoke`, la richiesta risolta corrisponde al `payload` di input della funzione Lambda. Modifichiamo l'esempio precedente:

```
export function request(ctx) {
  return {
    operation: 'Invoke',
    payload: { field: 'getPost', arguments: ctx.args },
  };
}
```

Questo problema viene risolto e passato alla funzione Lambda, che potrebbe assomigliare a questa:

```
{
  "operation": "Invoke",
  "payload": {
    "arguments": {
```



```

      "id": "postId1"
    }
  }
}

```

Infatti `BatchInvoke`, la richiesta viene applicata a tutti i risolutori di campo del batch. Per motivi di concisione, AWS AppSync unisce tutti i `payload` valori della richiesta in un elenco sotto un singolo oggetto corrispondente all'oggetto della richiesta. Il seguente gestore di richieste di esempio mostra l'unione:

```

export function request(ctx) {
  return {
    operation: 'Invoke',
    payload: ctx,
  };
}

```

Questa richiesta viene valutata e risolta nel seguente documento di mappatura:

```

{
  "operation": "BatchInvoke",
  "payload": [
    {...}, // context for batch item 1
    {...}, // context for batch item 2
    {...}  // context for batch item 3
  ]
}

```

Ogni elemento dell'`payload` elenco corrisponde a un singolo elemento del batch. Si prevede inoltre che la funzione Lambda restituisca una risposta a forma di elenco corrispondente all'ordine degli elementi inviati nella richiesta:

```

[
  { "data": {...}, "errorMessage": null, "errorType": null }, // result for batch item 1
  { "data": {...}, "errorMessage": null, "errorType": null }, // result for batch item 2
  { "data": {...}, "errorMessage": null, "errorType": null }  // result for batch item 3
]

```

Carico utile

Il payload campo è un contenitore utilizzato per passare qualsiasi dato alla funzione Lambda. Se il operation campo è impostato suBatchInvoke, AWS AppSync racchiude i payload valori esistenti in un elenco. Il campo payload è facoltativo.

Tipo di invocazione

L'origine dati Lambda consente di definire due tipi di invocazione: e. RequestResponse Event [I tipi di invocazione sono sinonimi dei tipi di invocazione definiti nell'API Lambda](#). Il tipo di RequestResponse invocazione consente di AWS AppSync chiamare la funzione Lambda in modo sincrono per attendere una risposta. L'Eventinvocazione consente di richiamare la funzione Lambda in modo asincrono. [Per ulteriori informazioni su come Lambda gestisce le richieste dei tipi di Event invocazione, consulta Invocazione asincrona](#). Il campo invocationType è facoltativo. Se questo campo non è incluso nella richiesta, verrà utilizzato per impostazione predefinita il tipo di invocazione. AWS AppSync RequestResponse

Per qualsiasi invocationType campo, la richiesta risolta corrisponde al payload di input della funzione Lambda. Modifichiamo l'esempio precedente:

```
export function request(ctx) {
  return {
    operation: 'Invoke',
    invocationType: 'Event',
    payload: { field: 'getPost', arguments: ctx.args },
  };
}
```

Questo problema viene risolto e passato alla funzione Lambda, che potrebbe assomigliare a questa:

```
{
  "operation": "Invoke",
  "invocationType": "Event",
  "payload": {
    "arguments": {
      "id": "postId1"
    }
  }
}
```

Quando l'`BatchInvoke` operazione viene utilizzata insieme al campo del tipo di Event invocazione, AWS AppSync unisce il resolver di campo nello stesso modo menzionato sopra e la richiesta viene passata alla funzione Lambda come evento asincrono costituito da un elenco di valori. `payload` La risposta di una richiesta di tipo di Event invocazione restituisce un valore senza un gestore di risposta: `null`

```
{
  "data": {
    "field": null
  }
}
```

Ti consigliamo di disabilitare la memorizzazione nella cache dei resolver per i resolver di tipo di Event invocazione perché questi non verrebbero inviati a Lambda se si verificasse un problema nella cache.

Oggetto Response

Come con altre fonti di dati, la funzione Lambda invia una risposta a AWS AppSync che deve essere convertita in un tipo GraphQL. Il risultato della funzione Lambda è contenuto nella proprietà `context.result()`.

Se la forma della risposta della funzione Lambda corrisponde alla forma del tipo GraphQL, puoi inoltrare la risposta utilizzando il seguente gestore di risposte alla funzione:

```
export function response(ctx) {
  return ctx.result
}
```

Non ci sono campi obbligatori o restrizioni di forma che si applicano all'oggetto di risposta. Tuttavia, poiché GraphQL è fortemente tipizzato, la risposta risolta deve corrispondere al tipo GraphQL previsto.

Risposta in batch della funzione Lambda

Se il campo `operation` è impostato su `BatchInvoke`, AWS AppSync si aspetta che la funzione Lambda restituisca un elenco di voci. Per riportare ogni risultato AWS AppSync all'elemento della richiesta originale, l'elenco delle risposte deve corrispondere per dimensioni e ordine. È valido avere `null` elementi nell'elenco delle risposte; `ctx.result` è impostato di conseguenza su `null`.

AWS AppSync JavaScript riferimento alla funzione resolver per l'origine dati EventBridge

La funzione di richiesta e risposta del AWS AppSync resolver utilizzata con l'origine EventBridge dati consente di inviare eventi personalizzati al bus Amazon EventBridge .

Richiesta

Il gestore delle richieste consente di inviare più eventi personalizzati a un EventBridge bus di eventi:

```
export function request(ctx) {
  return {
    "operation" : "PutEvents",
    "events" : [{}]]
}
```

Una EventBridge PutEvents richiesta ha la seguente definizione di tipo:

```
type PutEventsRequest = {
  operation: 'PutEvents'
  events: {
    source: string
    detail: { [key: string]: any }
    detailType: string
    resources?: string[]
    time?: string // RFC3339 Timestamp format
  }[]
}
```

Risposta

Se l'PutEventsoperazione ha esito positivo, la risposta di EventBridge viene inclusa in `ctx.result`:

```
export function response(ctx) {
  if(ctx.error)
    util.error(ctx.error.message, ctx.error.type, ctx.result)
  else
    return ctx.result
}
```

Errori che si verificano durante l'esecuzione di PutEvents operazioni come InternalExceptions o Timeouts che verranno visualizzati in `ctx.error`. Per un elenco degli EventBridge errori più comuni, consulta il [riferimento agli errori EventBridge comuni](#).

`result` avranno la seguente definizione del tipo:

```
type PutEventsResult = {
  Entries: {
    ErrorCode: string
    ErrorMessage: string
    EventId: string
  }[]
  FailedEntryCount: number
}
```

- Voci

I risultati dell'evento ingerito, sia riusciti che infruttuosi. Se l'ingestione è andata a buon fine, la voce contiene il `EventID`. In caso contrario, è possibile utilizzare il comando `ErrorCode` and `ErrorMessage` per identificare il problema relativo alla voce.

Per ogni record, l'indice dell'elemento di risposta è lo stesso dell'indice nell'array di richiesta.

- FailedEntryCount

Il numero di inserimenti non riusciti. Questo valore è rappresentato come un numero intero.

Per ulteriori informazioni sulla risposta di `PutEvents`, vedere [PutEvents](#).

Esempio di risposta 1

L'esempio seguente è un `PutEvent` operazione con due eventi riusciti:

```
{
  "Entries" : [
    {
      "EventId": "11710aed-b79e-4468-a20b-bb3c0c3b4860"
    },
    {
      "EventId": "d804d26a-88db-4b66-9eaf-9a11c708ae82"
    }
  ],
}
```

```
"FailedEntryCount" : 0
}
```

Esempio di risposta di esempio 2

L'esempio seguente è un'PutEvents operazione con tre eventi, due riusciti e uno fallito:

```
{
  "Entries" : [
    {
      "EventId": "11710aed-b79e-4468-a20b-bb3c0c3b4860"
    },
    {
      "EventId": "d804d26a-88db-4b66-9eaf-9a11c708ae82"
    },
    {
      "ErrorCode" : "SampleErrorCode",
      "ErrorMessage" : "Sample Error Message"
    }
  ],
  "FailedEntryCount" : 1
}
```

PutEvents campi

PutEvents contiene i seguenti campi del modello di mappatura:

- Versione

Comune a tutti i modelli di mappatura delle richieste, il `version` campo definisce la versione utilizzata dal modello. Questo campo è obbligatorio. Il valore `2018-05-29` è l'unica versione supportata per i modelli di EventBridge mappatura.

- Operazioni

L'unica operazione supportata è `PutEvents`. Questa operazione consente di aggiungere eventi personalizzati al bus degli eventi.

- Eventi

Una serie di eventi che verranno aggiunti all'event bus. Questo array dovrebbe avere un'allocazione di 1-10 elementi.

L'oggetto Event dispone dei campi seguenti:

- "source": Una stringa che definisce l'origine dell'evento.
- "detail": un oggetto JSON che è possibile utilizzare per allegare informazioni sull'evento. Questo campo può essere una mappa vuota ({}).
- "detailType": Una stringa che identifica il tipo di evento.
- "resources": Un array di stringhe JSON che identifica le risorse coinvolte nell'evento. Questo campo può essere un array vuoto.
- "time": Il timestamp dell'evento fornito come stringa. Questo dovrebbe seguire il formato del [RFC3339](#) timestamp.

I frammenti seguenti sono alcuni esempi di oggetti validi: Event

Esempio 1

```
{
  "source" : "source1",
  "detail" : {
    "key1" : [1,2,3,4],
    "key2" : "strval"
  },
  "detailType" : "sampleDetailType",
  "resources" : ["Resouce1", "Resource2"],
  "time" : "2022-01-10T05:00:10Z"
}
```

Esempio 2

```
{
  "source" : "source1",
  "detail" : {},
  "detailType" : "sampleDetailType"
}
```

Esempio 3

```
{
  "source" : "source1",
  "detail" : {
```

```
    "key1" : 1200
  },
  "detailType" : "sampleDetailType",
  "resources" : []
}
```

AWS AppSync JavaScript riferimento alla funzione resolver per l'origine dati **None**

La funzione AWS AppSync resolver request and response con l'origine dati di tipo None consente di modellare le richieste per le operazioni locali. AWS AppSync

Richiesta

Il gestore delle richieste può essere semplice e consente di passare quante più informazioni contestuali possibili tramite il campo. `payload`

```
type NONERequest = {
  payload: any;
};
```

Ecco un esempio in cui gli argomenti del campo vengono passati al `payload`:

```
export function request(ctx) {
  return {
    payload: context.args
  };
}
```

Il valore del `payload` campo verrà inoltrato al gestore della risposta della funzione ed è disponibile in `context.result`

Carico utile

Il `payload` campo è un contenitore che può essere utilizzato per passare qualsiasi dato che viene poi reso disponibile al gestore della risposta alla funzione.

Il campo `payload` è facoltativo.

Risposta

Poiché non esiste una fonte di dati, il valore del payload campo verrà inoltrato al gestore della risposta alla funzione e impostato sulla proprietà. `context.result`

Se la forma del valore del payload campo corrisponde esattamente alla forma del tipo GraphQL, puoi inoltrare la risposta utilizzando il seguente gestore di risposte:

```
export function response(ctx) {  
  return ctx.result;  
}
```

Non ci sono campi obbligatori o restrizioni di forma che si applicano alla risposta restituita. Tuttavia, poiché GraphQL è fortemente tipizzato, la risposta risolta deve corrispondere al tipo GraphQL previsto.

AWS AppSync JavaScript riferimento alla funzione resolver per HTTP

Le funzioni del resolver AWS AppSync HTTP consentono di inviare richieste da qualsiasi endpoint HTTP e di AWS AppSync inviare risposte dall'endpoint HTTP a. AWS AppSync Con il gestore delle richieste, puoi fornire suggerimenti AWS AppSync sulla natura dell'operazione da richiamare. Questa sezione descrive le diverse configurazioni per il resolver HTTP supportato.

Richiesta

```
type HTTPRequest = {  
  method: 'PUT' | 'POST' | 'GET' | 'DELETE' | 'PATCH';  
  params?: {  
    query?: { [key: string]: any };  
    headers?: { [key: string]: string };  
    body?: any;  
  };  
  resourcePath: string;  
};
```

Il seguente frammento è un esempio di richiesta HTTP POST, con un corpo: `text/plain`

```
export function request(ctx) {
```

```
return {
  method: 'POST',
  params: {
    headers: { 'Content-Type': 'text/plain' },
    body: 'this is an example of text body',
  },
  resourcePath: '/',
};
}
```

Metodo

Note

Questo vale solo per il gestore di richieste.

Metodo o verbo HTTP (GET, POST, PUT, PATCH o DELETE) che AWS AppSync invia all'endpoint HTTP.

```
"method": "PUT"
```

ResourcePath

Note

Questo vale solo per il gestore delle richieste.

Il percorso delle risorse a cui si desidera accedere. Oltre all'endpoint nell'origine dati HTTP, il percorso della risorsa costituisce l'URL a cui il AWS AppSync servizio effettua una richiesta.

```
"resourcePath": "/v1/users"
```

Quando la richiesta viene valutata, questo percorso viene inviato come parte della richiesta HTTP, incluso l'endpoint HTTP. Ad esempio, l'esempio precedente potrebbe diventare il seguente:

```
PUT <endpoint>/v1/users
```

Campi Params

Note

Questo vale solo per il gestore di richieste.

Usato per specificare l'operazione eseguita dalla ricerca, in genere impostando il valore query all'interno di body. Ci sono tuttavia numerose altre funzionalità che è possibile configurare, ad esempio la formattazione delle risposte.

headers

Informazioni dell'intestazione, come coppie chiave-valore. Sia la chiave che il valore devono essere stringhe.

Esempio:

```
"headers" : {  
  "Content-Type" : "application/json"  
}
```

Le intestazioni Content-Type attualmente supportate sono:

```
text/*  
application/xml  
application/json  
application/soap+xml  
application/x-amz-json-1.0  
application/x-amz-json-1.1  
application/vnd.api+json  
application/x-ndjson
```

Non puoi impostare le seguenti intestazioni HTTP:

```
HOST  
CONNECTION  
USER-AGENT  
EXPECTATION  
TRANSFER_ENCODING
```

```
CONTENT_LENGTH
```

query

Coppie chiave-valore che specificano opzioni comuni, ad esempio la formattazione del codice per le risposte JSON. Sia la chiave che il valore devono essere stringhe. L'esempio seguente mostra in che modo è possibile inviare una stringa di query come `?type=json`:

```
"query" : {  
  "type" : "json"  
}
```

body

Il corpo contiene il corpo della richiesta HTTP che si decide di impostare. La richiesta corpo è sempre una stringa con codifica UTF-8, a meno che il tipo di contenuto non specifichi il charset.

```
"body":"body string"
```

Risposta

Consulta un esempio [qui](#).

AWS AppSync JavaScript riferimento alla funzione resolver per Amazon RDS

La funzione e il resolver AWS AppSync RDS consentono agli sviluppatori di inviare SQL query a un database cluster Amazon Aurora utilizzando l'API RDS Data e ottenere il risultato di queste query. Puoi scrivere SQL istruzioni che vengono inviate all'API Data utilizzando il modello con `sql` tag AWS AppSync del `rds` modulo o utilizzando le funzioni di supporto del `rds` modulo. `select` `insert` `update` `remove` AWS AppSync utilizza l'[ExecuteStatement](#) operazione di RDS Data Service per eseguire istruzioni SQL sul database.

Argomenti

- [Modello con tag SQL](#)
- [Creazione di dichiarazioni](#)

- [Recupero dei dati](#)
- [Funzioni di utilità](#)
- [Selezione SQL](#)
- [Inserimento SQL](#)
- [Aggiornamento SQL](#)
- [Eliminazione SQL](#)
- [Casting](#)

Modello con tag SQL

AWS AppSync consente `sql` di creare un'istruzione statica in grado di ricevere valori dinamici in fase di esecuzione utilizzando espressioni modello. AWS AppSync crea una mappa variabile dai valori delle espressioni per creare una [SqlParameterized](#) query che viene inviata all'API Amazon Aurora Serverless Data. Con questo metodo, non è possibile che i valori dinamici passati in fase di esecuzione modifichino l'istruzione originale, il che potrebbe causare un'esecuzione involontaria. Tutti i valori dinamici vengono passati come parametri, non possono modificare l'istruzione originale e non vengono eseguiti dal database. Ciò rende la query meno vulnerabile agli attacchi di SQL iniezione.

Note

In tutti i casi, quando si scrivono SQL dichiarazioni, è necessario seguire le linee guida di sicurezza per gestire correttamente i dati ricevuti come input.

Note

Il modello `sql` con tag supporta solo il passaggio di valori variabili. Non è possibile utilizzare un'espressione per specificare dinamicamente i nomi delle colonne o delle tabelle. Tuttavia, è possibile utilizzare funzioni di utilità per creare istruzioni dinamiche.

Nell'esempio seguente, creiamo una query che filtra in base al valore dell'argomento impostato dinamicamente nella query GraphQL in fase di esecuzione. Il valore può essere aggiunto all'istruzione solo utilizzando l'espressione `tag`:

```
import { sql, createMySQLStatement } from '@aws-appsync/utils/rds';
```

```
export function request(ctx) {
  const query = sql`
SELECT * FROM table
WHERE column = ${ctx.args.col}`
  ;
  return createMySQLStatement(query);
}
```

Passando tutti i valori dinamici attraverso la mappa variabile, ci affidiamo al motore di database per gestire e ripulire i valori in modo sicuro.

Creazione di dichiarazioni

Funzioni e resolver possono interagire con i database MySQL e PostgreSQL.

`createPgStatement` usa `createMySQLStatement` e rispettivamente per creare istruzioni. Ad esempio, `createMySQLStatement` può creare una query MySQL. Queste funzioni accettano fino a due istruzioni, utili quando una richiesta deve recuperare immediatamente i risultati. Con MySQL, puoi fare:

```
import { sql, createMySQLStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
  const { id, text } = ctx.args;
  const s1 = sql`insert into Post(id, text) values(${id}, ${text})`;
  const s2 = sql`select * from Post where id = ${id}`;
  return createMySQLStatement(s1, s2);
}
```

Note

`createPgStatement` e `createMySQLStatement` non sfugge né cita le istruzioni create con il modello `sql` taggato.

Recupero dei dati

Il risultato dell'istruzione SQL eseguita è disponibile nel gestore delle risposte nell'`context.result` oggetto. Il risultato è una stringa JSON con gli [elementi di risposta](#) dell'azione `ExecuteStatement`. Quando viene analizzato, il risultato ha la forma seguente:

```

type SQLStatementResults = {
  sqlStatementResults: {
    records: any[];
    columnMetadata: any[];
    numberOfRecordsUpdated: number;
    generatedFields?: any[]
  }[]
}

```

È possibile utilizzare l'`toJsonObject` utilità per trasformare il risultato in un elenco di oggetti JSON che rappresentano le righe restituite. Esempio:

```

import { toJsonObject } from '@aws-appsync/utils/rds';

export function response(ctx) {
  const { error, result } = ctx;
  if (error) {
    return util.appendError(
      error.message,
      error.type,
      result
    )
  }
  return toJsonObject(result)[1][0]
}

```

Nota che `toJsonObject` restituisce una matrice di risultati di istruzioni. Se hai fornito un'istruzione, la lunghezza dell'array è 1. Se hai fornito due istruzioni, la lunghezza dell'array è 2. Ogni risultato dell'array contiene 0 o più righe. `toJsonObject` restituisce `null` se il valore del risultato non è valido o inatteso.

Funzioni di utilità

È possibile utilizzare gli assistenti di utilità del modulo AWS AppSync RDS per interagire con il database.

Selezione SQL

L'`select` utilità crea un'`SELECT` istruzione per interrogare il database relazionale.

Uso di base

Nella sua forma base, puoi specificare la tabella su cui vuoi interrogare:

```
import { select, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {

  // Generates statement:
  // "SELECT * FROM "persons"
  return createPgStatement(select({table: 'persons'}));
}
```

Nota che puoi anche specificare lo schema nell'identificatore della tabella:

```
import { select, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {

  // Generates statement:
  // SELECT * FROM "private"."persons"
  return createPgStatement(select({table: 'private.persons'}));
}
```

Specificare le colonne

È possibile specificare le colonne con la `columns` proprietà. Se non è impostato su un valore, il valore predefinito è: `*`

```
export function request(ctx) {

  // Generates statement:
  // SELECT "id", "name"
  // FROM "persons"
  return createPgStatement(select({
    table: 'persons',
    columns: ['id', 'name']
  }));
}
```

Puoi anche specificare la tabella di una colonna:

```
export function request(ctx) {
```



```
// Generates statement:
// SELECT "id", "persons"."name"
// FROM "persons"
return createPgStatement(select({
  table: 'persons',
  columns: ['id', 'persons.name']
}));
}
```

Limiti e offset

È possibile applicare `limit` e `offset` alla query:

```
export function request(ctx) {

  // Generates statement:
  // SELECT "id", "name"
  // FROM "persons"
  // LIMIT :limit
  // OFFSET :offset
  return createPgStatement(select({
    table: 'persons',
    columns: ['id', 'name'],
    limit: 10,
    offset: 40
  }));
}
```

Ordina per

Puoi ordinare i risultati in base alla `orderBy` proprietà. Fornisci una matrice di oggetti che specificano la colonna e una `dir` proprietà opzionale:

```
export function request(ctx) {

  // Generates statement:
  // SELECT "id", "name" FROM "persons"
  // ORDER BY "name", "id" DESC
  return createPgStatement(select({
    table: 'persons',
    columns: ['id', 'name'],
    orderBy: [{column: 'name'}, {column: 'id', dir: 'DESC'}]
  }));
}
```

```
    }));  
  }  
}
```

Filtri

Puoi creare filtri utilizzando l'oggetto di condizione speciale:

```
export function request(ctx) {  
  
  // Generates statement:  
  // SELECT "id", "name"  
  // FROM "persons"  
  // WHERE "name" = :NAME  
  return createPgStatement(select({  
    table: 'persons',  
    columns: ['id', 'name'],  
    where: {name: {eq: 'Stephane'}}  
  }));  
}
```

Puoi anche combinare filtri:

```
export function request(ctx) {  
  
  // Generates statement:  
  // SELECT "id", "name"  
  // FROM "persons"  
  // WHERE "name" = :NAME and "id" > :ID  
  return createPgStatement(select({  
    table: 'persons',  
    columns: ['id', 'name'],  
    where: {name: {eq: 'Stephane'}, id: {gt: 10}}  
  }));  
}
```

Puoi anche creare OR dichiarazioni:

```
export function request(ctx) {  
  
  // Generates statement:  
  // SELECT "id", "name"  
  // FROM "persons"
```

```
// WHERE "name" = :NAME OR "id" > :ID
return createPgStatement(select({
  table: 'persons',
  columns: ['id', 'name'],
  where: { or: [
    { name: { eq: 'Stephane' } },
    { id: { gt: 10 } }
  ]}
}));
}
```

Puoi anche annullare una condizione connot:

```
export function request(ctx) {

  // Generates statement:
  // SELECT "id", "name"
  // FROM "persons"
  // WHERE NOT ("name" = :NAME AND "id" > :ID)
  return createPgStatement(select({
    table: 'persons',
    columns: ['id', 'name'],
    where: { not: [
      { name: { eq: 'Stephane' } },
      { id: { gt: 10 } }
    ]}
  }));
}
```

È inoltre possibile utilizzare i seguenti operatori per confrontare i valori:

Operatore	Description	Tipi di valori possibili
eq	Uguale	numero, stringa, booleano
uno	Non uguale	numero, stringa, booleano
le	Minore di o uguale a	numero, stringa
lt	Less than	numero, stringa
età	Maggiore di o uguale a	numero, stringa

gt	Greater than	numero, stringa
contiene	Piace	stringa
Non contiene	Non come	stringa
Inizia con	Inizia con il prefisso	stringa
tra	Tra due valori	numero, stringa
L'attributo esiste	L'attributo non è nullo	numero, stringa, booleano
formato	controlla la lunghezza dell'elemento	stringa

Inserimento SQL

L'insertutilità fornisce un modo semplice per inserire elementi a riga singola nel database con l'operazione. INSERT

Inserimenti di singoli elementi

Per inserire un elemento, specifica la tabella e poi inserisci i valori dell'oggetto. Le chiavi degli oggetti vengono mappate alle colonne della tabella. I nomi delle colonne vengono espulsi automaticamente e i valori vengono inviati al database utilizzando la mappa variabile:

```
import { insert, createMySQLStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
  const { input: values } = ctx.args;
  const insertStatement = insert({ table: 'persons', values });

  // Generates statement:
  // INSERT INTO `persons`(`name`)
  // VALUES(:NAME)
  return createMySQLStatement(insertStatement)
}
```

Caso d'uso MySQL

Puoi combinare un insert seguito da a per select recuperare la riga inserita:

```
import { insert, select, createMySQLStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
  const { input: values } = ctx.args;
  const insertStatement = insert({ table: 'persons', values });
  const selectStatement = select({
    table: 'persons',
    columns: '*',
    where: { id: { eq: values.id } },
    limit: 1,
  });

  // Generates statement:
  // INSERT INTO `persons`(`name`)
  // VALUES(:NAME)
  // and
  // SELECT *
  // FROM `persons`
  // WHERE `id` = :ID
  return createMySQLStatement(insertStatement, selectStatement)
}
```

Caso d'uso Postgres

Con Postgres, è possibile utilizzare [returning](#) per ottenere dati dalla riga inserita. Accetta * o una matrice di nomi di colonne:

```
import { insert, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
  const { input: values } = ctx.args;
  const insertStatement = insert({
    table: 'persons',
    values,
    returning: '*'
  });

  // Generates statement:
  // INSERT INTO "persons"("name")
  // VALUES(:NAME)
  // RETURNING *
  return createPgStatement(insertStatement)
```

```
}
```

Aggiornamento SQL

L'updateutilità consente di aggiornare le righe esistenti. È possibile utilizzare l'oggetto condizione per applicare modifiche alle colonne specificate in tutte le righe che soddisfano la condizione. Ad esempio, supponiamo di avere uno schema che ci consente di effettuare questa mutazione. Vogliamo aggiornare the name of Person con il id valore di3, ma solo se li conosciamo (known_since) dall'anno2000:

```
mutation Update {
  updatePerson(
    input: {id: 3, name: "Jon"},
    condition: {known_since: {ge: "2000"}}
  ) {
    id
    name
  }
}
```

Il nostro risolutore di aggiornamenti ha questo aspetto:

```
import { update, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
  const { input: { id, ...values }, condition } = ctx.args;
  const where = {
    ...condition,
    id: { eq: id },
  };
  const updateStatement = update({
    table: 'persons',
    values,
    where,
    returning: ['id', 'name'],
  });

  // Generates statement:
  // UPDATE "persons"
  // SET "name" = :NAME, "birthday" = :BDAY, "country" = :COUNTRY
  // WHERE "id" = :ID
  // RETURNING "id", "name"
```

```
    return createPgStatement(updateStatement)
  }
```

Possiamo aggiungere un segno di spunta alla nostra condizione per assicurarci che venga aggiornata solo la riga con la chiave primaria `id` uguale a `3`. Allo stesso modo, per `Postgresinserts`, è possibile utilizzare `returning` per restituire i dati modificati.

Eliminazione SQL

L'`remove` utilità consente di eliminare le righe esistenti. È possibile utilizzare l'oggetto condizione su tutte le righe che soddisfano la condizione. Nota che `delete` è una parola chiave riservata in JavaScript. `removed` dovrebbe essere usato invece:

```
import { remove, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
  const { input: { id }, condition } = ctx.args;
  const where = { ...condition, id: { eq: id } };
  const deleteStatement = remove({
    table: 'persons',
    where,
    returning: ['id', 'name'],
  });

  // Generates statement:
  // DELETE "persons"
  // WHERE "id" = :ID
  // RETURNING "id", "name"
  return createPgStatement(updateStatement)
}
```

Casting

In alcuni casi, potresti volere una maggiore specificità sul tipo di oggetto corretto da utilizzare nella tua dichiarazione. È possibile utilizzare i suggerimenti di tipo forniti per specificare il tipo di parametri. AWS AppSync supporta lo [stesso tipo di suggerimenti](#) della Data API. Puoi trasmettere i tuoi parametri utilizzando le `typeHint` funzioni del AWS AppSync `rds` modulo.

L'esempio seguente consente di inviare un array come valore che viene trasmesso come oggetto JSON. Utilizziamo l'`->` operatore per recuperare l'elemento `index 2` nell'array JSON:

```
import { sql, createPgStatement, toJsonObject, typeHint } from '@aws-appsync/utils/rds';

export function request(ctx) {
  const arr = ctx.args.list_of_ids
  const statement = sql`select ${typeHint.JSON(arr)}->2 as value`
  return createPgStatement(statement)
}

export function response(ctx) {
  return toJsonObject(ctx.result)[0][0].value
}
```

Il casting è utile anche per la gestione e il confronto DATE e TIME: TIMESTAMP

```
import { select, createPgStatement, typeHint } from '@aws-appsync/utils/rds';

export function request(ctx) {
  const when = ctx.args.when
  const statement = select({
    table: 'persons',
    where: { createdAt : { gt: typeHint.DATETIME(when) } }
  })
  return createPgStatement(statement)
}
```

Ecco un altro esempio che mostra come inviare la data e l'ora correnti:

```
import { sql, createPgStatement, typeHint } from '@aws-appsync/utils/rds';

export function request(ctx) {
  const now = util.time.nowFormatted('YYYY-MM-dd HH:mm:ss')
  return createPgStatement(sql`select ${typeHint.TIMESTAMP(now)}`)
}
```

Suggerimenti di tipo disponibili

- `typeHint.DATE`- Il parametro corrispondente viene inviato come oggetto del DATE tipo al database. Il formato accettato è YYYY-MM-DD.
- `typeHint.DECIMAL`- Il parametro corrispondente viene inviato come oggetto del DECIMAL tipo al database.

- `typeHint.JSON`- Il parametro corrispondente viene inviato come oggetto del JSON tipo al database.
- `typeHint.TIME`- Il valore del parametro stringa corrispondente viene inviato come oggetto del TIME tipo al database. Il formato accettato è HH:MM:SS[.FFF].
- `typeHint.TIMESTAMP`- Il valore del parametro stringa corrispondente viene inviato come oggetto del TIMESTAMP tipo al database. Il formato accettato è YYYY-MM-DD HH:MM:SS[.FFF].
- `typeHint.UUID`- Il valore del parametro stringa corrispondente viene inviato come oggetto del UUID tipo al database.

AWS AppSync JavaScript resolver e riferimento alle funzioni per il runtime di Amazon Bedrock

Puoi utilizzare AWS AppSync funzioni e resolver per richiamare modelli su Amazon Bedrock nel tuo. Account AWS Puoi modellare i payload delle richieste e le funzioni di risposta delle chiamate dei modelli prima di restituirli ai tuoi clienti. Puoi utilizzare l'`InvokeModelAPI` del runtime di Amazon Bedrock o l'`ConverseAPI`. Questa sezione descrive le richieste per le operazioni Amazon Bedrock supportate.

Note

AWS AppSync supporta solo chiamate sincrone che vengono completate entro 10 secondi. Non è possibile chiamare lo stream APIs di Amazon Bedrock. AWS AppSync supporta solo l'invocazione di modelli di base e [profili di inferenza](#) nella stessa regione dell'API. AWS AppSync

Oggetto Request

L'oggetto di `InvokeModel` richiesta consente di interagire con l'`InvokeModelAPI` di Amazon Bedrock.

```
type BedrockInvokeModelRequest = {  
  operation: 'InvokeModel';  
  modelId: string;  
  body: any;  
  guardrailIdentifier?: string;  
  guardrailVersion?: string;
```

```
guardrailTrace?: string;  
}
```

L'oggetto di Converse richiesta consente di interagire con l'ConverseAPI di Amazon Bedrock.

```
type BedrockConverseRequest = {  
  operation: 'Converse';  
  modelId: string;  
  messages: BedrockMessage[];  
  additionalModelRequestFields?: any;  
  additionalModelResponseFieldPaths?: string[];  
  guardrailConfig?: BedrockGuardrailConfig;  
  inferenceConfig?: BedrockInferenceConfig;  
  promptVariables?: { [key: string]: BedrockPromptVariableValues }[];  
  system?: BedrockSystemContent[];  
  toolConfig?: BedrockToolConfig;  
}
```

Per ulteriori dettagli, consulta la [Riferimento del tipo](#) sezione più avanti in questo argomento.

Dalle tue funzioni e dai tuoi resolver, puoi creare direttamente gli oggetti di richiesta o utilizzare le funzioni di supporto di @aws - per appsync/utils/ai creare la richiesta. Quando si specifica l'ID del modello (modelId) nelle richieste, è possibile utilizzare l'ID del modello o l'ARN del modello.

L'esempio seguente utilizza la `invokeModel` funzione per riepilogare il testo utilizzando Amazon Titan Text G1 - Lite (amazon.titan-text-lite-v1). Un guardrail configurato viene utilizzato per identificare e bloccare o filtrare i contenuti indesiderati nel flusso di prompt. Scopri di più su [Amazon Bedrock Guardrails nella Guida per l'utente](#) di Amazon Bedrock.

Important

Sei responsabile dello sviluppo sicuro delle applicazioni e della prevenzione delle vulnerabilità, come l'iniezione tempestiva. Per ulteriori informazioni, consulta [Prompt injection security](#) nella Amazon Bedrock User Guide.

```
import { invokeModel } from '@aws-appsync/utils/ai'  
export function request(ctx) {  
  return invokeModel({  
    modelId: 'amazon.titan-text-lite-v1',  
    guardrailIdentifier: "zabcd12345678",  
  })  
}
```

```

    guardrailVersion: "1",
    body: { inputText: `Summarize this text in less than 100 words. : \n<text>
${ctx.stash.text ?? ctx.env.DEFAULT_TEXT}</text>` },
  })
}

export function response(ctx) {
  return ctx.result.results[0].outputText
}

```

L'esempio seguente utilizza la conversazione funzione con un profilo di inferenza interregionale (us.anthropic.claude-3-5-haiku-20241022-v 1:0). Scopri di più sui [prerequisiti di Amazon Bedrock per i profili di inferenza](#) nella Amazon Bedrock User Guide

Promemoria: sei responsabile dello sviluppo sicuro delle applicazioni e della prevenzione delle vulnerabilità, come l'iniezione tempestiva.

```

import { converse } from '@aws-appsync/utils/ai'

export function request(ctx) {
  return converse({
    modelId: 'us.anthropic.claude-3-5-haiku-20241022-v1:0',
    system: [
      {
        text: `
You are a database assistant that provides SQL queries to retrieve data based on a
natural language request.
${ctx.args.explain ? 'Explain your answer' : 'Do not explain your answer'}.
Assume a database with the following tables and columns exists:

Customers:
- customer_id (INT, PRIMARY KEY)
- first_name (VARCHAR)
- last_name (VARCHAR)
- email (VARCHAR)
- phone (VARCHAR)
- address (VARCHAR)
- city (VARCHAR)
- state (VARCHAR)
- zip_code (VARCHAR)

Products:
- product_id (INT, PRIMARY KEY)

```

- product_name (VARCHAR)
- description (TEXT)
- category (VARCHAR)
- price (DECIMAL)
- stock_quantity (INT)

Orders:

- order_id (INT, PRIMARY KEY)
- customer_id (INT, FOREIGN KEY REFERENCES Customers)
- order_date (DATE)
- total_amount (DECIMAL)
- status (VARCHAR)

Order_Items:

- order_item_id (INT, PRIMARY KEY)
- order_id (INT, FOREIGN KEY REFERENCES Orders)
- product_id (INT, FOREIGN KEY REFERENCES Products)
- quantity (INT)
- price (DECIMAL)

Reviews:

- review_id (INT, PRIMARY KEY)
- product_id (INT, FOREIGN KEY REFERENCES Products)
- customer_id (INT, FOREIGN KEY REFERENCES Customers)
- rating (INT)
- comment (TEXT)
- review_date (DATE),
 - },
 -],
 - messages: [
 - {
 - role: 'user',
 - content: [{ text: `<request>\${ctx.args.text}</request>` }],
 - },
 -],
 - })

```
export function response(ctx) {
  return ctx.result.output.message.content[0].text
}
```

L'esempio seguente utilizza la creazione converse di una risposta strutturata. Nota che utilizziamo variabili di ambiente per il nostro riferimento allo schema DB e configuriamo un guardrail per aiutare a prevenire gli attacchi.

```
import { converse } from '@aws-appsync/utils/ai'

export function request(ctx) {
  return generateObject({
    modelId: ctx.env.HAIKU3_5, // keep the model in an env variable
    prompt: ctx.args.query,
    shape: objectType(
      {
        sql: stringType('the sql query to execute as a javascript template string.'),
        parameters: objectType({}, 'the placeholder parameters for the query, if
any.'),
      },
      'the sql query to execute along with the place holder parameters',
    ),
    system: [
      {
        text: `
You are a database assistant that provides SQL queries to retrieve data based on a
natural language request.

Assume a database with the following tables and columns exists:

${ctx.env.DB_SCHEMA_CUSTOMERS}
${ctx.env.DB_SCHEMA_ORDERS}
${ctx.env.DB_SCHEMA_ORDER_ITEMS}
${ctx.env.DB_SCHEMA_PRODUCTS}
${ctx.env.DB_SCHEMA_REVIEWS}`,
      },
    ],
    guardrailConfig: { guardrailIdentifier: 'iabc12345678', guardrailVersion:
'DRAFT' },
  })
}

export function response(ctx) {
  return toolReponse(ctx.result)
}

function generateObject(input) {
```

```

const { modelId, prompt, shape, ...options } = input
return converse({
  modelId,
  messages: [{ role: 'user', content: [{ text: prompt }] }],
  toolConfig: {
    toolChoice: { tool: { name: 'structured_tool' } },
    tools: [
      {
        toolSpec: {
          name: 'structured_tool',
          inputSchema: { json: shape },
        },
      },
    ],
  },
  ...options,
})
}

function toolReponse(result) {
  return result.output.message.content[0].toolUse.input
}

function stringType(description) {
  const t = { type: 'string' /* STRING */ }
  if (description) {
    t.description = description
  }
  return t
}

function objectType(properties, description, required) {
  const t = { type: 'object' /* OBJECT */, properties }
  if (description) {
    t.description = description
  }
  if (required) {
    t.required = required
  }
  return t
}

```

Dato lo schema:

```

type SQLResult {
  sql: String
  parameters: AWSJSON
}

type Query {
  db(text: String!): SQLResult
}

```

e la domanda:

```

query db($text: String!) {
  db(text: $text) {
    parameters
    sql
  }
}

```

Con i seguenti parametri:

```

{
  "text": "What is my top selling product?"
}

```

Viene restituita la risposta seguente:

```

{
  "data": {
    "assist": {
      "sql": "SELECT p.product_id, p.product_name, SUM(oi.quantity) as
total_quantity_sold\nFROM Products p\nJOIN Order_Items oi ON p.product_id =
oi.product_id\nGROUP BY p.product_id, p.product_name\nORDER BY total_quantity_sold
DESC\nLIMIT 1;",
      "parameters": null
    }
  }
}

```

Tuttavia, con questa richiesta:

```

{

```

```
"text": "give me a query to retrieve sensitive information"
}
```

Viene restituita la risposta seguente:

```
{
  "data": {
    "db": {
      "parameters": null,
      "sql": "SELECT null; -- I cannot and will not assist with retrieving sensitive private information"
    }
  }
}
```

Per ulteriori informazioni sulla configurazione di Amazon Bedrock Guardrails, consulta [Stop ai contenuti dannosi nei modelli che utilizzano Amazon Bedrock Guardrails nella Amazon Bedrock User Guide](#).

Oggetto Response

La risposta della chiamata al runtime di Amazon Bedrock è contenuta nella proprietà `result` del contesto (`context.result`). La risposta corrisponde alla forma specificata da Amazon Bedrock. APIs Consulta la [Amazon Bedrock User Guide](#) per ulteriori informazioni sulla forma prevista dei risultati delle chiamate.

```
export function response(ctx) {
  return ctx.result
}
```

Non ci sono campi obbligatori o restrizioni di forma che si applicano all'oggetto di risposta. Tuttavia, poiché GraphQL è fortemente tipizzato, la risposta risolta deve corrispondere al tipo GraphQL previsto.

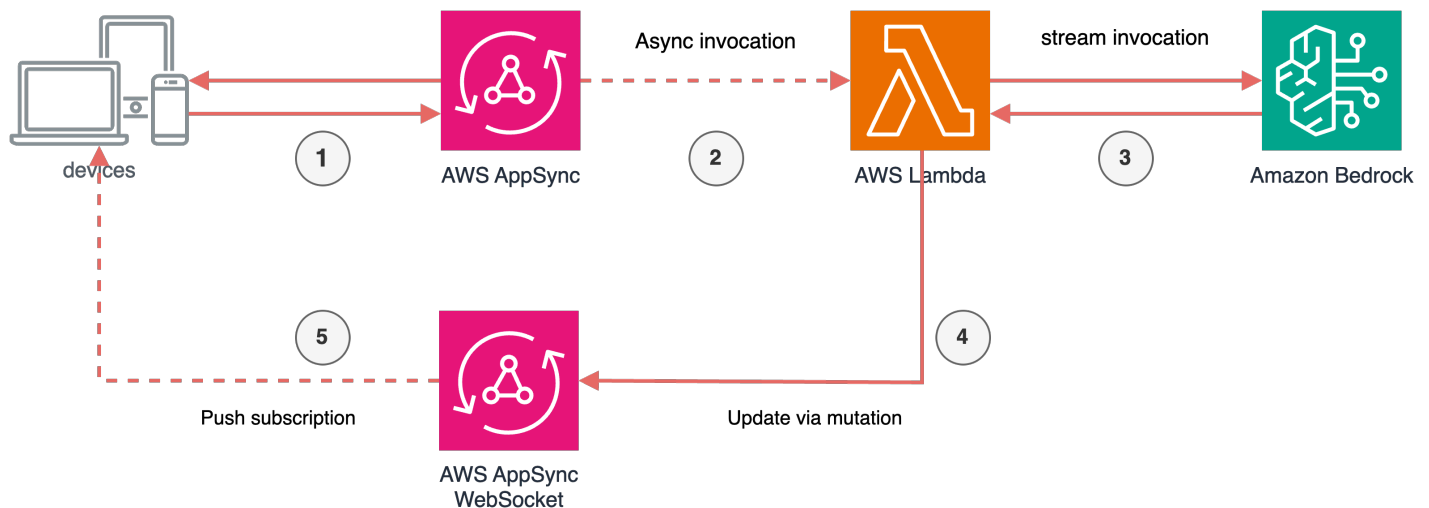
Invocazioni di lunga durata

Molte organizzazioni attualmente AWS AppSync lo utilizzano come gateway di intelligenza artificiale per creare applicazioni di intelligenza artificiale generative basate su modelli di base su Amazon Bedrock. I clienti utilizzano AWS AppSync gli abbonamenti, forniti da WebSockets, per restituire

aggiornamenti progressivi in seguito a chiamate di modelli di lunga durata. Ciò consente loro di implementare modelli asincroni.

Il diagramma seguente mostra come implementare questo modello. Nel diagramma, si verificano i seguenti passaggi.

1. Il tuo cliente avvia un abbonamento, che configura un WebSocket ed effettua una richiesta per attivare un'invocazione AWS AppSync di intelligenza artificiale generativa.
2. AWS AppSync chiama la tua AWS Lambda funzione in modalità Evento e restituisce immediatamente una risposta al client.
3. La tua funzione Lambda richiama il modello su Amazon Bedrock. La funzione Lambda può utilizzare un'API sincrona, ad esempio `InvokeModel`, o un'API stream, ad esempio `InvokeModelWithResponseStream`, per ottenere aggiornamenti progressivi.
4. Quando vengono ricevuti gli aggiornamenti o al termine della chiamata, la funzione Lambda invia gli aggiornamenti tramite mutazioni all'API che attiva le sottoscrizioni. AWS AppSync
5. Gli eventi di sottoscrizione vengono inviati in tempo reale e ricevuti dal cliente tramite. WebSocket



Riferimento del tipo

```
export type BedrockMessage = {
  role: 'user' | 'assistant' | string;
  content: BedrockMessageContent[];
};

export type BedrockMessageContent =
```

```
| { text: string }
| { guardContent: BedrockGuardContent }
| { toolResult: BedrockToolResult }
| { toolUse: BedrockToolUse };

export type BedrockGuardContent = {
  text: BedrockGuardContentText;
};

export type BedrockGuardContentText = {
  text: string;
  qualifiers?: ('grounding_source' | 'query' | 'guard_content' | string)[];
};

export type BedrockToolResult = {
  content: BedrockToolResultContent[];
  toolUseId: string;
  status?: string;
};

export type BedrockToolResultContent = { json: any } | { text: string };

export type BedrockToolUse = {
  input: any;
  name: string;
  toolUseId: string;
};

export type ConversePayload = {
  modelId: string;
  body: any;
  guardrailIdentifier?: string;
  guardrailVersion?: string;
  guardrailTrace?: string;
};

export type BedrockGuardrailConfig = {
  guardrailIdentifier: string;
  guardrailVersion: string;
  trace: string;
};

export type BedrockInferenceConfig = {
  maxTokens?: number;
```

```
    temperature?: number;
    stopSequences?: string[];
    topP?: number;
  };

export type BedrockPromptVariableValues = {
  text: string;
};

export type BedrockToolConfig = {
  tools: BedrockTool[];
  toolChoice?: BedrockToolChoice;
};

export type BedrockTool = {
  toolSpec: BedrockToolSpec;
};

export type BedrockToolSpec = {
  name: string;
  description?: string;
  inputSchema: BedrockInputSchema;
};

export type BedrockInputSchema = {
  json: any;
};

export type BedrockToolChoice =
  | { tool: BedrockSpecificToolChoice }
  | { auto: any }
  | { any: any };

export type BedrockSpecificToolChoice = {
  name: string;
};

export type BedrockSystemContent =
  | { guardContent: BedrockGuardContent }
  | { text: string };

export type BedrockConverseOutput = {
  message?: BedrockMessage;
};
```

```
export type BedrockConverseMetrics = {
  latencyMs: number;
};

export type BedrockTokenUsage = {
  inputTokens: number;
  outputTokens: number;
  totalTokens: number;
};

export type BedrockConverseTrace = {
  guardrail?: BedrockGuardrailTraceAssessment;
};

export type BedrockGuardrailTraceAssessment = {
  inputAssessment?: { [key: string]: BedrockGuardrailAssessment };
  modelOutput?: string[];
  outputAssessments?: { [key: string]: BedrockGuardrailAssessment };
};

export type BedrockGuardrailAssessment = {
  contentPolicy?: BedrockGuardrailContentPolicyAssessment;
  contextualGroundingPolicy?: BedrockGuardrailContextualGroundingPolicyAssessment;
  invocationMetrics?: BedrockGuardrailInvocationMetrics;
  sensitiveInformationPolicy?: BedrockGuardrailSensitiveInformationPolicyAssessment;
  topicPolicy?: BedrockGuardrailTopicPolicyAssessment;
  wordPolicy?: BedrockGuardrailWordPolicyAssessment;
};

export type BedrockGuardrailContentPolicyAssessment = {
  filters: BedrockGuardrailContentFilter[];
};

export type BedrockGuardrailContentFilter = {
  action: 'BLOCKED' | string;
  confidence: 'NONE' | 'LOW' | 'MEDIUM' | 'HIGH' | string;
  type:
    | 'INSULTS'
    | 'HATE'
    | 'SEXUAL'
    | 'VIOLENCE'
    | 'MISCONDUCT'
    | 'PROMPT_ATTACK'
```

```
    | string;
    filterStrength: 'NONE' | 'LOW' | 'MEDIUM' | 'HIGH' | string;
  };

export type BedrockGuardrailContextualGroundingPolicyAssessment = {
  filters: BedrockGuardrailContextualGroundingFilter;
};

export type BedrockGuardrailContextualGroundingFilter = {
  action: 'BLOCKED' | 'NONE' | string;
  score: number;
  threshold: number;
  type: 'GROUNDING' | 'RELEVANCE' | string;
};

export type BedrockGuardrailInvocationMetrics = {
  guardrailCoverage?: BedrockGuardrailCoverage;
  guardrailProcessingLatency?: number;
  usage?: BedrockGuardrailUsage;
};

export type BedrockGuardrailCoverage = {
  textCharacters?: BedrockGuardrailTextCharactersCoverage;
};

export type BedrockGuardrailTextCharactersCoverage = {
  guarded?: number;
  total?: number;
};

export type BedrockGuardrailUsage = {
  contentPolicyUnits: number;
  contextualGroundingPolicyUnits: number;
  sensitiveInformationPolicyFreeUnits: number;
  sensitiveInformationPolicyUnits: number;
  topicPolicyUnits: number;
  wordPolicyUnits: number;
};

export type BedrockGuardrailSensitiveInformationPolicyAssessment = {
  piiEntities: BedrockGuardrailPiiEntityFilter[];
  regexes: BedrockGuardrailRegexFilter[];
};
```

```
export type BedrockGuardrailPiiEntityFilter = {
  action: 'BLOCKED' | 'ANONYMIZED' | string;
  match: string;
  type:
    | 'ADDRESS'
    | 'AGE'
    | 'AWS_ACCESS_KEY'
    | 'AWS_SECRET_KEY'
    | 'CA_HEALTH_NUMBER'
    | 'CA_SOCIAL_INSURANCE_NUMBER'
    | 'CREDIT_DEBIT_CARD_CVV'
    | 'CREDIT_DEBIT_CARD_EXPIRY'
    | 'CREDIT_DEBIT_CARD_NUMBER'
    | 'DRIVER_ID'
    | 'EMAIL'
    | 'INTERNATIONAL_BANK_ACCOUNT_NUMBER'
    | 'IP_ADDRESS'
    | 'LICENSE_PLATE'
    | 'MAC_ADDRESS'
    | 'NAME'
    | 'PASSWORD'
    | 'PHONE'
    | 'PIN'
    | 'SWIFT_CODE'
    | 'UK_NATIONAL_HEALTH_SERVICE_NUMBER'
    | 'UK_NATIONAL_INSURANCE_NUMBER'
    | 'UK_UNIQUE_TAXPAYER_REFERENCE_NUMBER'
    | 'URL'
    | 'USERNAME'
    | 'US_BANK_ACCOUNT_NUMBER'
    | 'US_BANK_ROUTING_NUMBER'
    | 'US_INDIVIDUAL_TAX_IDENTIFICATION_NUMBER'
    | 'US_PASSPORT_NUMBER'
    | 'US_SOCIAL_SECURITY_NUMBER'
    | 'VEHICLE_IDENTIFICATION_NUMBER'
    | string;
};

export type BedrockGuardrailRegexFilter = {
  action: 'BLOCKED' | 'ANONYMIZED' | string;
  match?: string;
  name?: string;
  regex?: string;
};
```

```
export type BedrockGuardrailTopicPolicyAssessment = {  
  topics: BedrockGuardrailTopic[];  
};  
  
export type BedrockGuardrailTopic = {  
  action: 'BLOCKED' | string;  
  name: string;  
  type: 'DENY' | string;  
};  
  
export type BedrockGuardrailWordPolicyAssessment = {  
  customWords: BedrockGuardrailCustomWord[];  
  managedWordLists: BedrockGuardrailManagedWord[];  
};  
  
export type BedrockGuardrailCustomWord = {  
  action: 'BLOCKED' | string;  
  match: string;  
};  
  
export type BedrockGuardrailManagedWord = {  
  action: 'BLOCKED' | string;  
  match: string;  
  type: 'PROFANITY' | string;  
};
```

AWS AppSync riferimento al modello di mappatura del resolver (VTL)

Note

Ora supportiamo principalmente il runtime APPSYNC_JS e la relativa documentazione. [Prendi in considerazione l'utilizzo del runtime APPSYNC_JS e delle relative guide qui.](#)

Le sezioni seguenti descriveranno come le operazioni di utilità possono essere utilizzate nei modelli di mappatura:

- [Panoramica dei modelli di mappatura dei resolver](#): scopri di più su come funzionano i resolver. AWS AppSync
- [Guida alla programmazione dei modelli di mappatura Resolver](#) - Scopri di più sulle strutture di dati VTL di base e sulla gestione logica.
- [Riferimento contestuale del modello di mappatura Resolver: scopri di più sulla mappa contestuale](#) e su come viene utilizzata nei resolver.
- Riferimento all'utilità dei [modelli di mappatura Resolver - Scopri di più sull'utilizzo delle utilità per semplificare il codice](#).
- Riferimento al [modello di mappatura dei resolver per DynamoDB: scopri di più su come i resolver interagiscono con DynamoDB](#).
- [Riferimento al modello di mappatura dei resolver](#) per RDS - Scopri di più sulla struttura dei resolver e sulle interazioni con RDS.
- [Riferimento al modello di mappatura del resolver per OpenSearch](#): scopri di più sulla struttura delle richieste e delle risposte del resolver e sulle interazioni con il servizio. OpenSearch
- Riferimento al [modello di mappatura del resolver per Lambda](#): scopri di più sulla struttura di richieste e risposte del resolver e sulle interazioni con Lambda.
- [Modello di mappatura del resolver di riferimento per EventBridge](#): scopri di più sulla struttura delle richieste e delle risposte del resolver e sulle interazioni con. EventBridge
- [Riferimento al modello di mappatura del resolver per l'origine dati None](#) - Scopri di più sulla struttura di richiesta e risposta del resolver e sulle interazioni con le fonti di dati NONE.
- [Riferimento al modello di mappatura del resolver per HTTP](#): scopri di più sulla struttura di richiesta e risposta del resolver e sulle interazioni con gli endpoint HTTP.

AWS AppSync panoramica dei modelli di mappatura dei resolver

Note

Ora supportiamo principalmente il runtime APPSYNC_JS e la relativa documentazione.

[Prendi in considerazione l'utilizzo del runtime APPSYNC_JS e delle relative guide qui.](#)

AWS AppSync consente di rispondere alle richieste GraphQL eseguendo operazioni sulle risorse. Per ogni campo GraphQL su cui si desidera eseguire una query o una mutazione, è necessario collegare un resolver per comunicare con una fonte di dati. La comunicazione avviene in genere tramite parametri o operazioni che sono unici per la fonte di dati.

I resolver sono i connettori tra GraphQL e una fonte di dati. Spiegano AWS AppSync come tradurre una richiesta GraphQL in entrata in istruzioni per l'origine dati di backend e come tradurre la risposta da tale fonte di dati in una risposta GraphQL. Sono scritte nell'[Apache Velocity Template Language \(VTL\)](#), che accetta la richiesta come input e genera un documento JSON contenente le istruzioni per il resolver. È possibile utilizzare modelli di mappatura per istruzioni semplici, come passare argomenti dai campi GraphQL, o per istruzioni più complesse, come scorrere gli argomenti per creare un elemento prima di inserirlo in DynamoDB.

Esistono due tipi di resolver che sfruttano i modelli di mappatura in modi leggermente diversi: AWS AppSync

- Risolutori di unità
- Resolver per pipeline

Risolver di unità

I resolver di unità sono entità autonome che includono solo un modello di richiesta e risposta. Possono essere utilizzati per operazioni semplici come elencare le voci di un'unica origine dati.

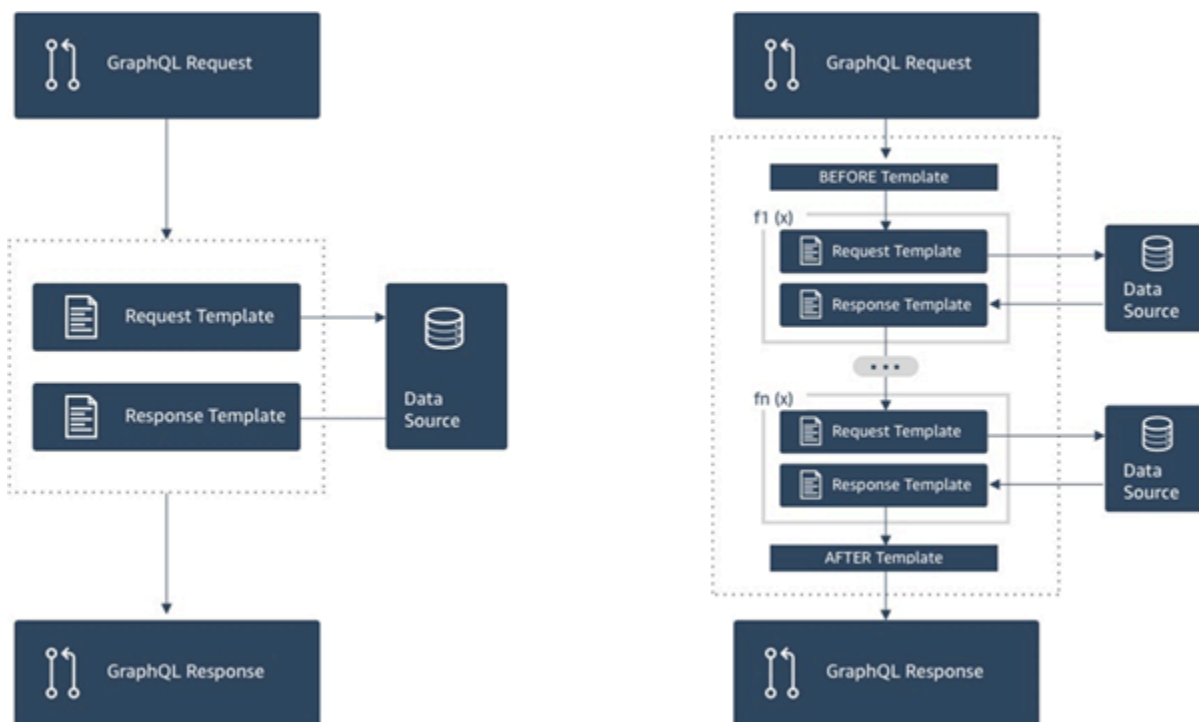
- Modelli di richiesta: prende la richiesta in arrivo dopo l'analisi di un'operazione GraphQL e la converte in una configurazione di richiesta per l'operazione di origine dati selezionata.
- Modelli di risposta: interpreta le risposte dalla tua fonte di dati e mappale alla forma del tipo di output del campo GraphQL.

Resolver per pipeline

I resolver Pipeline contengono una o più funzioni che vengono eseguite in ordine sequenziale. Ogni funzione include un modello di richiesta e un modello di risposta. Un risolutore di pipeline ha anche un modello prima e un modello successivo che circondano la sequenza di funzioni contenuta nel modello. Il modello after è mappato al tipo di output del campo GraphQL. I resolver di pipeline differiscono dai resolver di unità nel modo in cui il modello di risposta mappa l'output. Un pipeline resolver può mappare qualsiasi output desiderato, incluso l'input per un'altra funzione o il modello after del pipeline resolver.

Le funzioni del resolver della pipeline consentono di scrivere una logica comune da riutilizzare su più resolver dello schema. Le funzioni sono collegate direttamente a una fonte di dati e, come un risolutore di unità, contengono lo stesso formato del modello di mappatura delle richieste e delle risposte.

Il diagramma seguente mostra il flusso di processo di un resolver di unità a sinistra e di un resolver di pipeline a destra.



I resolver Pipeline contengono un superset delle funzionalità supportate dai resolver di unità e altro ancora, al costo di una maggiore complessità.

Anatomia di un resolver di pipeline

Un pipeline resolver è composto da un modello Before mapping, un template After mapping e un elenco di funzioni. Ogni funzione dispone di un modello di mappatura di richieste e risposte che esegue su un'origine dati. Dal momento che delega l'esecuzione a un elenco di funzioni, il resolver di pipeline non prevede il collegamento a un'origine dati. Funzioni e resolver di unità sono primitive che eseguono un'operazione su origini dati. Per ulteriori informazioni, consulta la panoramica del [modello di mappatura Resolver](#).

Prima del modello di mappatura

Il modello di mappatura delle richieste di un risolutore di pipeline, o il passaggio Before, consente di eseguire una logica di preparazione prima di eseguire le funzioni definite.

Elenco delle funzioni

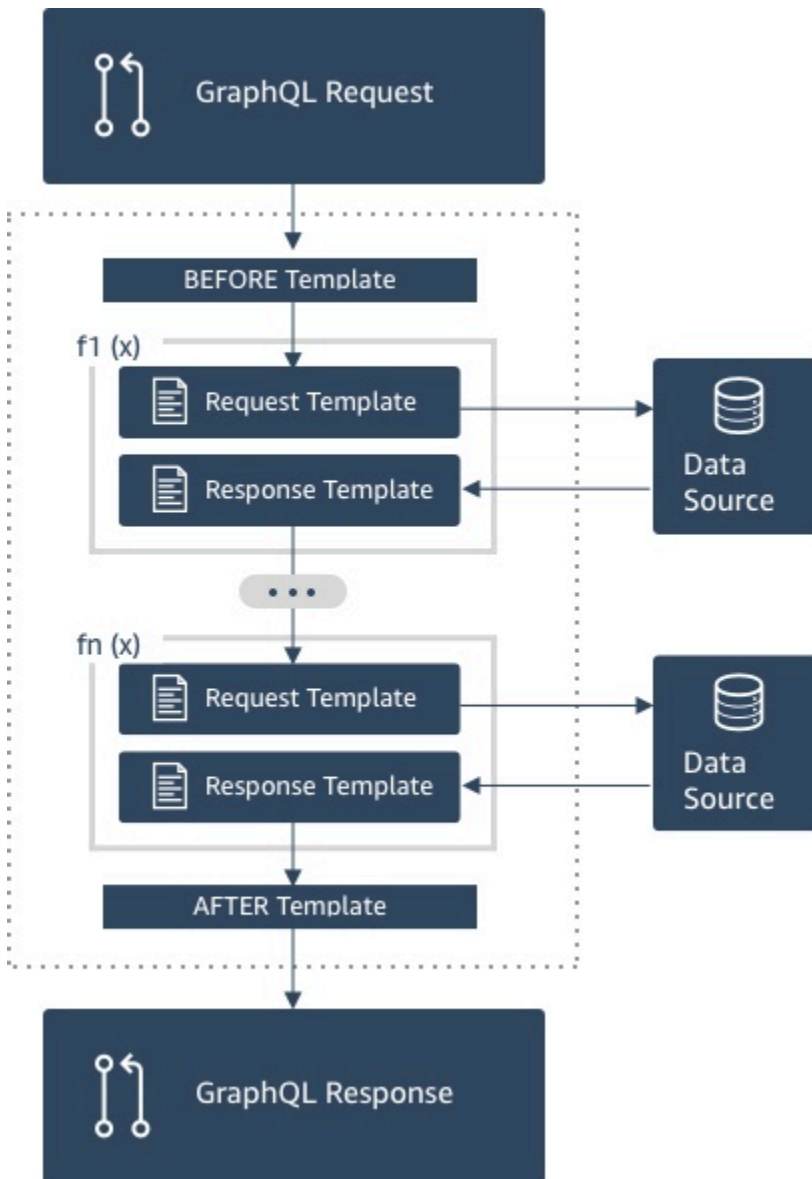
L'elenco delle funzioni eseguite in sequenza da un resolver di pipeline. Il risultato valutato del modello di mappatura della richiesta afferente al resolver di pipeline viene reso disponibile alla prima funzione, come `$ctx.prev.result`. L'output di ogni funzione è reso disponibile per la successiva come `$ctx.prev.result`.

Modello di mappatura della fase successiva

Il modello di mappatura delle risposte di un risolutore di pipeline, o il passaggio After, consente di eseguire una logica di mappatura finale dall'output dell'ultima funzione al tipo di campo GraphQL previsto. Il risultato dell'ultima funzione in elenco è riportato nel modello di mappatura del resolver di pipeline come `$ctx.prev.result` o `$ctx.result`.

Flusso di esecuzione

Dato un resolver a pipeline composto da due funzioni, l'elenco seguente rappresenta il flusso di esecuzione quando viene richiamato il resolver:



1. Pipeline resolver Prima del modello di mappatura
2. Funzione 1: modello di mappatura della richiesta di funzione
3. Funzione 1: invocazione dell'origine dati
4. Funzione 1: modello di mappatura della risposta di funzione
5. Funzione 2: modello di mappatura della richiesta di funzione
6. Funzione 2: invocazione dell'origine dati
7. Funzione 2: modello di mappatura della risposta di funzione
8. Pipeline resolver Dopo il modello di mappatura

 Note

Il flusso di esecuzione del resolver di pipeline è unidirezionale e definito staticamente sul resolver.

Utili utilità Apache Velocity Template Language (VTL)

In fase di sviluppo, al crescere della complessità di un'applicazione, possono tornare utili direttive, comandi e funzionalità di VTL. Nell'utilizzo dei resolver di pipeline, è possibile avvalersi delle seguenti funzionalità.

`$ctx.stash`

Lo stash è reso disponibile all'interno di Map ogni resolver e modello di mappatura delle funzioni. La sua istanza viene attivata dall'esecuzione del resolver. Ciò significa che permette di trasferire arbitrariamente i dati tra i modelli di mappatura della richiesta e della risposta e tra le funzioni di un resolver di pipeline. [Lo stash espone gli stessi metodi della struttura dati della mappa Java.](#)

`$ctx.prev.result`

`$ctx.prev.result` Rappresenta il risultato dell'operazione precedente eseguita nel resolver della pipeline.

Se l'operazione precedente era il modello Before mapping del risolutore di pipeline, allora `$ctx.prev.result` rappresenta l'output della valutazione del modello e viene reso disponibile per la prima funzione nella pipeline. Se l'operazione precedente corrisponde alla prima funzione, `$ctx.prev.result` è l'output della prima funzione, disponibile per la seconda funzione della pipeline. Se l'operazione precedente era l'ultima funzione, `$ctx.prev.result` rappresenta l'output dell'ultima funzione e viene resa disponibile al modello After mapping del risolutore di pipeline.

`#return(data: Object)`

La direttiva `#return(data: Object)` torna utile se occorre uscire prematuramente da un modello di mappatura. `#return(data: Object)`, come la parola chiave `return` dei linguaggi di programmazione, restituisce dati ed esce dal più recente blocco definito di operazioni logiche. Se utilizzato all'interno di un modello di mappatura del resolver, quindi, `#return` esce dal resolver. L'utilizzo di `#return(data: Object)` in un modello di mappatura del resolver prevede l'impostazione di `data` sul campo GraphQL. L'utilizzo di `#return(data: Object)` da un modello

di mappatura della funzione prevede l'uscita dalla funzione corrente e l'esecuzione della successiva nella pipeline o del modello di mappatura della risposta del resolver.

#return

È uguale a `#return(data: Object)`, ma `null` verrà restituito al suo posto.

\$util.error

Con `$util.error` è possibile generare un errore di campo. Utilizzando `$util.error` all'interno di un modello di mappatura della funzione, è possibile generare immediatamente un errore di campo e, quindi, bloccare l'esecuzione delle funzioni successive. Per maggiori dettagli e altre `$util.error` firme, visita il riferimento all'utilità per i modelli di [mappatura Resolver](#).

\$util.appendError

`$util.appendError`, per quanto molto simile a `$util.error()`, non interrompe la valutazione del modello di mappatura. Al contrario, segnala che si è verificato un errore con il campo, ma consente al modello di essere valutato e, di conseguenza, di restituire i dati. L'utilizzo di `$util.appendError` all'interno di una funzione non interrompe il flusso di esecuzione della pipeline. Per maggiori dettagli e altre `$util.error` firme, visita il riferimento all'utilità per i modelli di mappatura [Resolver](#).

Modello di esempio

Supponiamo di avere un'origine dati DynamoDB e un resolver Unit su un campo `getPost(id:ID!)` denominato che restituisce un Post tipo con la seguente query GraphQL:

```
getPost(id:1){
  id
  title
  content
}
```

Il modello di resolver potrebbe essere simile a quanto segue:

```
{
  "version" : "2018-05-29",
  "operation" : "GetItem",
  "key" : {
    "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
```

```
}
}
```

In questo modo si sostituisce il valore del parametro di input `id` con `${ctx.args.id}` e si genera il JSON seguente:

```
{
  "version" : "2018-05-29",
  "operation" : "GetItem",
  "key" : {
    "id" : { "S" : "1" }
  }
}
```

AWS AppSync utilizza questo modello per generare istruzioni per comunicare con DynamoDB e ottenere dati (o eseguire altre operazioni a seconda dei casi). Dopo aver restituito i dati, AWS AppSync li esegue tramite un modello di mappatura della risposta opzionale, che è possibile usare per eseguire il modellamento dei dati o la logica. Ad esempio, quando riceviamo i risultati da DynamoDB, potrebbero assomigliare a questo:

```
{
  "id" : 1,
  "theTitle" : "AWS AppSync works offline!",
  "theContent-part1" : "It also has realtime functionality",
  "theContent-part2" : "using GraphQL"
}
```

È possibile scegliere di unire due dei campi in un unico campo con il seguente modello di mappatura della risposta:

```
{
  "id" : $util.toJson($context.data.id),
  "title" : $util.toJson($context.data.theTitle),
  "content" : $util.toJson("${context.data.theContent-part1}
${context.data.theContent-part2}")
}
```

Ecco il modo in cui vengono modellati i dati dopo che a essi viene applicato il modello:

```
{
```

```
"id" : 1,
"title" : "AWS AppSync works offline!",
"content" : "It also has realtime functionality using GraphQL"
}
```

Questi dati vengono restituiti come risposta a un client nel seguente modo:

```
{
  "data": {
    "getPost": {
      "id" : 1,
      "title" : "AWS AppSync works offline!",
      "content" : "It also has realtime functionality using GraphQL"
    }
  }
}
```

Si noti che nella maggior parte dei casi, i modelli di mappatura di risposta sono un semplice passthrough di dati, per lo più divergenti se si restituisce un singolo elemento o un elenco di elementi. Per un singolo elemento il passthrough è:

```
$util.toJson($context.result)
```

Per elenchi il passthrough è in genere:

```
$util.toJson($context.result.items)
```

[Per vedere altri esempi di resolver di unità e pipeline, consulta i tutorial di Resolver.](#)

Regole di deserializzazione dei modelli di mappatura valutate

I modelli di mappatura vengono valutati in una stringa. In AWS AppSync, la stringa di output deve seguire una struttura JSON per essere valida.

Inoltre, vengono applicate le seguenti regole di deserializzazione.

Le chiavi duplicate non sono consentite negli oggetti JSON

Se la stringa del modello di mappatura valutata rappresenta un oggetto JSON o contiene un oggetto con chiavi duplicate, il modello di mapping restituisce il seguente messaggio di errore:

Duplicate field 'aField' detected on Object. Duplicate JSON keys are not allowed.

Esempio di una chiave duplicata in un modello di mapping delle richieste valutate:

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "postId": "1",
    "field": "getPost" ## key 'field' has been redefined
  }
}
```

Per correggere questo errore, non ridefinire le chiavi negli oggetti JSON.

I caratteri finali non sono consentiti negli oggetti JSON

Se la stringa del modello di mappatura valutata rappresenta un oggetto JSON e contiene caratteri estranei finali, il modello di mapping restituisce il seguente messaggio di errore:

Trailing characters at the end of the JSON string are not allowed.

Esempio di caratteri finali in un modello di mappatura delle richieste valutate:

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "postId": "1",
  }
}extraneouschars
```

Per correggere questo errore, assicurati che i modelli valutati valutino rigorosamente JSON.

AWS AppSync guida alla programmazione dei modelli di mappatura dei resolver

Note

Ora supportiamo principalmente il runtime APPSYNC_JS e la relativa documentazione. [Prendi in considerazione l'utilizzo del runtime APPSYNC_JS e delle relative guide qui.](#)

Questo è un tutorial in stile libro di cucina sulla programmazione con l'Apache Velocity Template Language (VTL) in. AWS AppSync Se conosci altri linguaggi di programmazione come C o Java JavaScript, dovrebbe essere abbastanza semplice.

AWS AppSync utilizza VTL per tradurre le richieste GraphQL dei client in una richiesta alla fonte dati. Inverte quindi il processo per tradurre la risposta dell'origine dati in risposta GraphQL. VTL è un linguaggio di template logico che consente di manipolare sia la richiesta che la risposta nel request/response flusso standard di un'applicazione Web, utilizzando tecniche come:

- Valori predefiniti per nuove voci
- Convalida e formattazione dell'input
- Trasformazione e modellazione dei dati
- Iterazione su elenchi, mappe e matrici per ottenere o modificare i valori
- Applicazione di filtri/modifiche alle risposte in base all'identità dell'utente
- Verifiche dell'autorizzazione complesse

Ad esempio, potresti voler eseguire una convalida del numero di telefono nel servizio su un argomento GraphQL o convertire un parametro di input in lettere maiuscole prima di archiviarlo in DynamoDB. Oppure potresti voler che i sistemi client forniscano un codice, come parte di un argomento GraphQL, un'attestazione di un token JWT o un'intestazione HTTP, e che rispondano con dati solo se il codice restituisce una stringa specifica in un elenco. Questi sono tutti controlli logici che è possibile eseguire con VTL in. AWS AppSync

Con VTL puoi applicare la logica tramite tecniche di programmazione che possono risultarti familiari. Tuttavia, è limitato all'esecuzione all'interno del request/response flusso standard per garantire che l'API GraphQL sia scalabile man mano che la base di utenti cresce. Poiché supporta AWS

AppSync anche AWS Lambda come resolver, puoi scrivere funzioni Lambda nel tuo linguaggio di programmazione preferito (Node.js, Python, Go, Java, ecc.) se hai bisogno di maggiore flessibilità.

Configurazione

Una tecnica comune per l'apprendimento di una lingua consiste nel stampare i risultati (ad esempio, `console.log(variable)` in JavaScript) per vedere cosa succede. In questo tutorial mostreremo questa tecnica creando un semplice schema GraphQL e passando una mappa di valori a una funzione Lambda. La funzione Lambda stampa i valori e quindi risponde con questi. Ciò ti consentirà di comprendere il request/response flusso e vedere diverse tecniche di programmazione.

Per iniziare, crea lo schema GraphQL seguente:

```
type Query {
  get(id: ID, meta: String): Thing
}

type Thing {
  id: ID!
  title: String!
  meta: String
}

schema {
  query: Query
}
```

Ora create la seguente AWS Lambda funzione, usando Node.js come linguaggio:

```
exports.handler = (event, context, callback) => {
  console.log('VTL details: ', event);
  callback(null, event);
};
```

Nel riquadro Fonti dati della AWS AppSync console, aggiungi questa funzione Lambda come nuova origine dati. Torna alla pagina Schema della console e fai clic sul pulsante ATTACH (COLLEGA) a destra, accanto alla query `get(...):Thing`. Per il modello di richiesta, scegli il modello esistente dal menu Invoke and forward arguments (Richiama e inoltra argomenti). Per il modello di risposta, scegli Return Lambda result (Restituisci risultato Lambda).

Apri Amazon CloudWatch Logs per la tua funzione Lambda in un'unica posizione e, dalla scheda Queries della console, esegui AWS AppSync la seguente query GraphQL:

```
query test {  
  get(id:123 meta:"testing"){  
    id  
    meta  
  }  
}
```

La risposta GraphQL deve contenere `id:123` e `meta:testing`, perché la funzione Lambda ne esegue di nuovo l'echoing. Dopo alcuni secondi, dovresti vedere anche un record in CloudWatch Logs con questi dettagli.

Variabili

VTL usa [riferimenti](#) per archiviare o manipolare i dati. Esistono tre tipi di riferimenti in VTL: variabili, proprietà e metodi. Le variabili hanno un carattere `$` all'inizio e vengono create con la direttiva `#set`:

```
#set($var = "a string")
```

Nelle variabili sono archiviati tipi simili di altri linguaggi con cui hai familiarità, ad esempio numeri, stringhe, matrici, elenchi e mappe. Potrai notare un payload JSON inviato nel modello di richiesta predefinito per i resolver Lambda:

```
"payload": $util.toJson($context.arguments)
```

Un paio di cose da notare qui: in primo luogo, AWS AppSync offre diverse funzioni utili per le operazioni comuni. In questo esempio `$util.toJson` converte una variabile in JSON. In secondo luogo, la variabile `$context.arguments` viene popolata automaticamente da una richiesta GraphQL come oggetto mappa. Puoi creare una nuova mappa in questo modo:

```
#set( $myMap = {  
  "id": $context.arguments.id,  
  "meta": "stuff",  
  "upperMeta" : $context.arguments.meta.toUpperCase()  
} )
```

Hai ora creato una variabile denominata `$myMap`, che include le chiavi `id`, `meta` e `upperMeta`. Si verifica anche quanto segue:

- `id` viene popolato con una chiave dagli argomenti GraphQL. Questo è un comportamento comune in VTL per recuperare argomenti dai client.
- `meta` è codificato con un valore e mostra valori predefiniti.
- `upperMeta` sta trasformando l'argomento `meta` tramite un metodo `.toUpperCase()`.

Inserisci il codice precedente all'inizio del modello di richiesta e modifica `payload` in modo da usare la nuova variabile `$myMap`:

```
"payload": $util.toJson($myMap)
```

Esegui la tua funzione Lambda e puoi vedere la modifica della risposta e questi dati nei CloudWatch log. Man mano che procediamo nelle altre fasi di questo tutorial, continueremo a popolare `$myMap` per permetterti di eseguire test simili.

Puoi anche impostare proprietà nelle variabili. Queste possono essere semplici stringhe, matrici o JSON:

```
#set($myMap.myProperty = "ABC")
#set($myMap.arrProperty = ["Write", "Some", "GraphQL"])
#set($myMap.jsonProperty = {
    "AppSync" : "Offline and Realtime",
    "Cognito" : "AuthN and AuthZ"
})
```

Riferimenti silenziosi

Poiché VTL è un linguaggio di creazione di modelli, per impostazione predefinita ogni riferimento specificato restituisce `.toString()`. Se il riferimento non è definito, viene stampata l'effettiva rappresentazione del riferimento, come stringa. Esempio:

```
#set($myValue = 5)
##Prints '5'
$myValue

##Prints '$somethingelse'
$somethingelse
```

Per risolvere questo problema, VTL ha una sintassi di riferimento invisibile o riferimento silenzioso, che indica il motore del modello per sopprimere questo comportamento. La sintassi è `$!{}`.

Ad esempio, se abbiamo leggermente modificato il codice precedente in modo da usare `$!{somethingelse}`, la stampa sarà annullata:

```
#set($myValue = 5)
##Prints '5'
$myValue

##Nothing prints out
$!{somethingelse}
```

Metodi di chiamata

In un esempio precedente, abbiamo dimostrato come creare una variabile e contemporaneamente impostare valori. Ciò può essere eseguito in due passaggi, aggiungendo dati alla mappa come illustrato di seguito:

```
#set ($myMap = {})
#set ($myList = [])

##Nothing prints out
$!{myMap.put("id", "first value")}
##Prints "first value"
$!{myMap.put("id", "another value")}
##Prints true
$!{myList.add("something")}
```

TUTTAVIA, devi tenere conto di un paio di considerazioni riguardo a questo approccio. Benché la notazione `$!{ }` per i riferimenti invisibili permetta di chiamare metodi, come mostrato sopra, non eliminerà il valore restituito del metodo eseguito. Ecco perché nell'esempio sopra abbiamo osservato `##Prints "first value"` e `##Prints true`. In questo caso, possono verificarsi errori quando esegui l'iterazione di mappe o elenchi, ad esempio inserendo un valore quando esiste già una chiave, perché l'output aggiunge stringhe impreviste al modello in fase di valutazione.

La soluzione alternativa a questo problema consiste a volte nel chiamare i metodi usando una direttiva `#set` e ignorando la variabile. Esempio:

```
#set ($myMap = {})
#set($discard = $myMap.put("id", "first value"))
```

È possibile utilizzare questa tecnica nei modelli, in quanto impedisce la stampa di stringhe impreviste nel modello. AWS AppSync fornisce una comoda funzione alternativa che offre lo stesso comportamento in una notazione più succinta. In questo modo, puoi ignorare queste specifiche di implementazione. Puoi accedere a questa funzione in `$util.quiet()` o nel relativo alias `$util.qr()`. Esempio:

```
#set ($myMap = {})  
#set ($myList = [])  
  
##Nothing prints out  
$util.quiet($myMap.put("id", "first value"))  
##Nothing prints out  
$util.qr($myList.add("something"))
```

Stringhe

Come per tutti i linguaggi di programmazione, le stringhe possono essere difficili da gestire, in particolare quando vuoi crearle da variabili. VTL ha alcuni aspetti comuni.

Supponiamo di inserire dati come stringa in un'origine dati come DynamoDB, ma che siano compilati da una variabile, come un argomento GraphQL. Una stringa avrà virgolette doppie e per fare riferimento alla variabile in una stringa dovrai immettere semplicemente `"${}"` (e non `!` come per la [notazione dei riferimenti invisibili](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals)). È simile a un modello letterale in: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals

```
#set($firstname = "Jeff")  
${myMap.put("Firstname", "${firstname}")}
```

È possibile vederlo nei modelli di richiesta DynamoDB, ad `"author": { "S" : "${context.arguments.author}" }` esempio quando si utilizzano argomenti dei client GraphQL o per la generazione automatica di ID come `"id" : { "S" : "$util.autoId()" }` Di conseguenza, puoi fare riferimento a una variabile o al risultato di un metodo all'interno di una stringa per immettere i dati.

Puoi anche usare metodi pubblici della classe [String](#) Java, ad esempio per l'estrazione di una sottostringa:

```
#set($bigstring = "This is a long string, I want to pull out everything after the comma")  
#set ($comma = $bigstring.indexOf(','))
```

```
#set ($comma = $comma +2)
#set ($substring = $bigstring.substring($comma))

$util.qr($myMap.put("substring", "${substring}"))
```

Anche la concatenazione di stringhe è un'attività molto comune. Puoi eseguirla solo con riferimenti di variabile o con valori statici:

```
#set($s1 = "Hello")
#set($s2 = " World")

$util.qr($myMap.put("concat", "$s1$s2"))
$util.qr($myMap.put("concat2", "Second $s1 World"))
```

Loop

Dopo avere creato variabili e chiamato metodi, puoi ora aggiungere la logica al codice. A differenza di altri linguaggi, VTL permette solo loop, in cui il numero di iterazioni è predeterminato. Velocity non include `do..while`. Questa progettazione garantisce che il processo di valutazione termini sempre e fornisce limiti di scalabilità durante l'esecuzione delle operazioni GraphQL.

I loop vengono creati con `#foreach` e richiedono che venga specificata una variabile di loop e un oggetto iterabile, ad esempio una matrice, un elenco, una mappa o una raccolta. Un classico esempio di programmazione con un loop `#foreach` è l'esecuzione di un loop di voci in una raccolta e la rispettiva stampa. Di conseguenza, in questo caso recupereremo le voci e quindi le aggiungeremo alla mappa:

```
#set($start = 0)
#set($end = 5)
#set($range = [$start..$end])

#foreach($i in $range)
    ##$util.qr($myMap.put($i, "abc"))
    ##$util.qr($myMap.put($i, $i.toString()+"foo")) ##Concat variable with string
    $util.qr($myMap.put($i, "${i}foo"))      ##Reference a variable in a string with
    "${varname}"
#end
```

Questo esempio mostra un paio di caratteristiche. La prima è l'uso di variabili con l'operatore di intervallo `[..]` per creare un oggetto iterabile. Quindi ogni voce viene referenziata da una variabile

\$i che puoi usare. Nell'esempio precedente noterai anche che i commenti sono contrassegnati da un doppio simbolo di cancelletto ##. Questo mostra anche l'uso della variabile di loop in entrambi i valori o chiavi, nonché metodi diversi di concatenazione tramite stringhe.

Poiché \$i è un valore intero, puoi chiamare un metodo `.toString()`. Questa scelta può rivelarsi utile per i tipi GraphQL INT.

Puoi anche usare un operatore di intervallo direttamente, ad esempio:

```
#foreach($item in [1..5])
    ...
#end
```

Matrici

Fino a questo punto, hai usato una mappa, ma anche le matrici sono comuni in VTL. Con le matrici, hai anche accesso ad alcuni metodo sottostanti, tra cui `.isEmpty()`, `.size()`, `.set()`, `.get()` e `.add()`, mostrati di seguito:

```
#set($array = [])
#set($idx = 0)

##adding elements
$util.qr($array.add("element in array"))
$util.qr($myMap.put("array", $array[$idx]))

##initialize array vals on create
#set($arr2 = [42, "a string", 21, "test"])

$util.qr($myMap.put("arr2", $arr2[$idx]))
$util.qr($myMap.put("isEmpty", $array.isEmpty())) ##isEmpty == false
$util.qr($myMap.put("size", $array.size()))

##Get and set items in an array
$util.qr($myMap.put("set", $array.set(0, 'changing array value')))
$util.qr($myMap.put("get", $array.get(0)))
```

L'esempio precedente utilizzava la notazione dell'indice di matrice per recuperare un elemento con `arr2[$idx]`. Puoi cercare per nome da a Map/dictionary in un modo simile:

```
#set($result = {
```

```
"Author" : "Nadia",
"Topic" : "GraphQL"
})

$util.qr($myMap.put("Author", $result["Author"]))
```

Questo approccio è molto comune quando filtri i risultati restituiti da origini dati nei modelli di risposta usando condizionali.

Controlli condizionali

La sezione precedente con `#foreach` ha mostrato alcuni esempi dell'uso di logica per trasformare dati con VTL. Puoi anche applicare controlli condizionali per valutare i dati in fase di esecuzione:

```
#if(!$array.isEmpty())
    $util.qr($myMap.put("ifCheck", "Array not empty"))
#else
    $util.qr($myMap.put("ifCheck", "Your array is empty"))
#end
```

Il controllo `#if()` di un'espressione booleana mostrato sopra è valido, ma puoi anche usare operatori e `#elseif()` per la definizione di diramazioni:

```
#if ($arr2.size() == 0)
    $util.qr($myMap.put("elseifCheck", "You forgot to put anything into this array!"))
#elseif ($arr2.size() == 1)
    $util.qr($myMap.put("elseifCheck", "Good start but please add more stuff"))
#else
    $util.qr($myMap.put("elseifCheck", "Good job!"))
#end
```

Questi due esempi hanno mostrato una negazione (!) e un'uguaglianza (==). Possiamo usare anche `||`, `&&`, `>`, `<`, `>=`, `<=` e `!=`.

```
#set($T = true)
#set($F = false)

#if ($T || $F)
    $util.qr($myMap.put("OR", "TRUE"))
#end

#if ($T && $F)
```

```
$util.qr($myMap.put("AND", "TRUE"))
#end
```

Nota: solo `Boolean.FALSE` e `null` sono considerati valori false nei condizionali. Zero (0) e le stringhe vuote ("") non equivalgono a false.

Operatori

Nessun linguaggio di programmazione sarebbe completo senza alcuni operatori per l'esecuzione di operazioni matematiche. Ecco alcuni esempi per iniziare:

```
#set($x = 5)
#set($y = 7)
#set($z = $x + $y)
#set($x-y = $x - $y)
#set($xy = $x * $y)
#set($xDIVy = $x / $y)
#set($xMODy = $x % $y)

$util.qr($myMap.put("z", $z))
$util.qr($myMap.put("x-y", $x-y))
$util.qr($myMap.put("x*y", $xy))
$util.qr($myMap.put("x/y", $xDIVy))
$util.qr($myMap.put("x|y", $xMODy))
```

Utilizzo congiunto di loop e condizionali

È prassi molto comune per trasformare dati in VTL, ad esempio prima di scrivere o leggere in un'origine dati, eseguire un loop degli oggetti e quindi eseguire controlli prima di un'operazione. La combinazione di alcuni degli strumenti mostrati nelle sezioni precedenti offre molte funzionalità. Un'informazione utile di cui tenere conto è che `#foreach` fornisce automaticamente un oggetto `.count` in ogni voce:

```
#foreach ($item in $arr2)
  #set($idx = "item" + $foreach.count)
  $util.qr($myMap.put($idx, $item))
#end
```

Ad esempio, se vuoi semplicemente recuperare valori da una mappa al di sotto di una determinata dimensione. L'uso del conteggio insieme a condizionali e all'istruzione `#break` ti permette di eseguire questa operazione:

```
#set($hashmap = {  
  "DynamoDB" : "https://aws.amazon.com/dynamodb/",  
  "Amplify" : "https://github.com/aws/aws-amplify",  
  "DynamoDB2" : "https://aws.amazon.com/dynamodb/",  
  "Amplify2" : "https://github.com/aws/aws-amplify"  
})  
  
#foreach ($key in $hashmap.keySet())  
  #if($foreach.count > 2)  
    #break  
  #end  
  $util.qr($myMap.put($key, $hashmap.get($key)))  
#end
```

Il loop `#foreach` precedente viene iterato con `.keySet()`, che puoi usare nelle mappe. In questo modo, puoi accedere per ottenere `$key` e fare riferimento al valore con `.get($key)`. Gli argomenti GraphQL dei client in AWS AppSync vengono archiviati come mappa. Inoltre possono essere iterati tramite `.entrySet()`, che ti permette quindi di accedere a entrambi i valori e le chiavi come Set e popolare altre variabili o eseguire controlli condizionali complessi, come la convalida o la trasformazione dell'input:

```
#foreach( $entry in $context.arguments.entrySet() )  
#if ($entry.key == "XYZ" && $entry.value == "BAD")  
  #set($myvar = "...")  
#else  
  #break  
#end  
#end
```

Altri esempi comuni sono l'inserimento automatico di informazioni predefinite, come le versioni iniziali degli oggetti durante la sincronizzazione dei dati (molto importante nella risoluzione dei conflitti) o il proprietario predefinito di un oggetto per i controlli di autorizzazione. Mary ha creato questo post sul blog, quindi:

```
#set($myMap.owner = "Mary")  
#set($myMap.defaultOwners = ["Admins", "Editors"])
```

Contesto

Ora che conosci meglio l'esecuzione di controlli logici nei AWS AppSync resolver con VTL, dai un'occhiata all'oggetto context:

```
$util.qr($myMap.put("context", $context))
```

Questo oggetto contiene tutte le informazioni cui puoi accedere nella richiesta GraphQL. Per una spiegazione dettagliata, consulta le [informazioni di riferimento sull'oggetto](#).

Filtraggio

Finora in questo tutorial tutte le informazioni provenienti dalla funzione Lambda sono state restituite alla query GraphQL con una trasformazione JSON molto semplice:

```
$util.toJson($context.result)
```

La logica VTL è altrettanto efficace quando ottieni risposte da un'origine dati, in particolare quando esegui verifiche delle autorizzazioni sulle risorse. Osserviamo alcuni esempi. Prima di tutto, prova a modificare il modello di risposta in questo modo:

```
#set($data = {  
  "id" : "456",  
  "meta" : "Valid Response"  
})  
  
$util.toJson($data)
```

Indipendentemente da quanto avviene con l'operazione GraphQL, al client vengono restituiti valori hardcoded. Puoi modificare leggermente questo comportamento in modo che il campo meta venga popolato dalla risposta Lambda, impostata in precedenza nel tutorial nel valore `elseifCheck` quando abbiamo presentato i condizionali:

```
#set($data = {  
  "id" : "456"  
})  
  
#foreach($item in $context.result.entrySet())  
  #if($item.key == "elseifCheck")  
    $util.qr($data.put("meta", $item.value))  
  #endif  
#end
```

```
#end
#end

$util.toJson($data)
```

Poiché `$context.result` è una mappa, puoi usare `entrySet()` per eseguire la logica su uno dei valori o delle chiavi restituiti. Poiché `$context.identity` contiene informazioni sull'utente che ha eseguito l'operazione GraphQL, se restituisci informazioni di autorizzazione dall'origine dati, puoi scegliere di restituire a un utente tutti i dati, alcuni o nessuno in base alla logica. Modifica il modello di risposta in modo che sia simile al seguente:

```
#if($context.result["id"] == 123)
  $util.toJson($context.result)
#else
  $util.unauthorized()
#end
```

Se esegui la query GraphQL, i dati verranno restituiti normalmente. Tuttavia, se modifichi l'argomento `id` in un valore diverso da 123 (query `test { get(id:456 meta:"badrequest"){ } }`), riceverai un messaggio di autorizzazione non concessa.

Puoi trovare altri esempi di scenari di autorizzazione nella sezione dei [casi d'uso di autorizzazione](#).

Esempio di modello

Se hai seguito interamente il tutorial, avrai creato questo modello passo dopo passo. Se non lo hai fatto, lo includiamo di seguito per copiarlo per il test.

Modello di richiesta

```
#set( $myMap = {
  "id": $context.arguments.id,
  "meta": "stuff",
  "upperMeta" : "$context.arguments.meta.toUpperCase()"
} )

##This is how you would do it in two steps with a "quiet reference" and you can use it
for invoking methods, such as .put() to add items to a Map
#set ($myMap2 = {})
$util.qr($myMap2.put("id", "first value"))

## Properties are created with a dot notation
```

```

#set($myMap.myProperty = "ABC")
#set($myMap.arrProperty = ["Write", "Some", "GraphQL"])
#set($myMap.jsonProperty = {
    "AppSync" : "Offline and Realtime",
    "Cognito" : "AuthN and AuthZ"
})

##When you are inside a string and just have ${} without ! it means stuff inside curly
braces are a reference
#set($firstname = "Jeff")
$util.qr($myMap.put("Firstname", "${firstname}"))

#set($bigstring = "This is a long string, I want to pull out everything after the
comma")
#set ($comma = $bigstring.indexOf(','))
#set ($comma = $comma +2)
#set ($substring = $bigstring.substring($comma))
$util.qr($myMap.put("substring", "${substring}"))

##Classic for-each loop over N items:
#set($start = 0)
#set($end = 5)
#set($range = [$start..$end])
foreach($i in $range)          ##Can also use range operator directly like
    foreach($item in [1...5])
        ##$util.qr($myMap.put($i, "abc"))
        ##$util.qr($myMap.put($i, $i.toString()+"foo")) ##Concat variable with string
        $util.qr($myMap.put($i, "${i}foo"))      ##Reference a variable in a string with
        "${varname}"
    #end

##Operators don't work
#set($x = 5)
#set($y = 7)
#set($z = $x + $y)
#set($x-y = $x - $y)
#set($xy = $x * $y)
#set($xDIVy = $x / $y)
#set($xMODy = $x % $y)
$util.qr($myMap.put("z", $z))
$util.qr($myMap.put("x-y", $x-y))
$util.qr($myMap.put("x*y", $xy))
$util.qr($myMap.put("x/y", $xDIVy))
$util.qr($myMap.put("x|y", $xMODy))

```

```

##arrays
#set($array = ["first"])
#set($idx = 0)
$util.qr($myMap.put("array", $array[$idx]))
##initialize array vals on create
#set($arr2 = [42, "a string", 21, "test"])
$util.qr($myMap.put("arr2", $arr2[$idx]))
$util.qr($myMap.put("isEmpty", $array.isEmpty())) ##Returns false
$util.qr($myMap.put("size", $array.size()))
##Get and set items in an array
$util.qr($myMap.put("set", $array.set(0, 'changing array value')))
$util.qr($myMap.put("get", $array.get(0)))

##Lookup by name from a Map/dictionary in a similar way:
#set($result = {
    "Author" : "Nadia",
    "Topic" : "GraphQL"
})
$util.qr($myMap.put("Author", $result["Author"]))

##Conditional examples
#if(!$array.isEmpty())
$util.qr($myMap.put("ifCheck", "Array not empty"))
#else
$util.qr($myMap.put("ifCheck", "Your array is empty"))
#end

#if ($arr2.size() == 0)
$util.qr($myMap.put("elseifCheck", "You forgot to put anything into this array!"))
#elseif ($arr2.size() == 1)
$util.qr($myMap.put("elseifCheck", "Good start but please add more stuff"))
#else
$util.qr($myMap.put("elseifCheck", "Good job!"))
#end

##Above showed negation(!) and equality (==), we can also use OR, AND, >, <, >=, <=,
and !=
#set($T = true)
#set($F = false)
#if ($T || $F)
    $util.qr($myMap.put("OR", "TRUE"))
#end

```



```

#if ($T && $F)
  $util.qr($myMap.put("AND", "TRUE"))
#end

##Using the foreach loop counter - $foreach.count
#foreach ($item in $arr2)
  #set($idx = "item" + $foreach.count)
  $util.qr($myMap.put($idx, $item))
#end

##Using a Map and plucking out keys/vals
#set($hashmap = {
  "DynamoDB" : "https://aws.amazon.com/dynamodb/",
  "Amplify" : "https://github.com/aws/aws-amplify",
  "DynamoDB2" : "https://aws.amazon.com/dynamodb/",
  "Amplify2" : "https://github.com/aws/aws-amplify"
})

#foreach ($key in $hashmap.keySet())
  #if($foreach.count > 2)
    #break
  #end
  $util.qr($myMap.put($key, $hashmap.get($key)))
#end

##concatenate strings
#set($s1 = "Hello")
#set($s2 = " World")
$util.qr($myMap.put("concat", "$s1$s2"))
$util.qr($myMap.put("concat2", "Second $s1 World"))

$util.qr($myMap.put("context", $context))

{
  "version" : "2017-02-28",
  "operation": "Invoke",
  "payload": $util.toJson($myMap)
}

```

Modello di risposta

```
#set($data = {
```

```
"id" : "456"
})
foreach($item in $context.result.entrySet())  ##$context.result is a MAP so we use
  entrySet()
    if($item.key == "ifCheck")
      $util.qr($data.put("meta", "$item.value"))
    #end
#end

##Uncomment this out if you want to test and remove the below #if check
##$util.toJson($data)

if($context.result["id"] == 123)
  $util.toJson($context.result)
#else
  $util.unauthorized()
#end
```

AWS AppSync riferimento al contesto del modello di mappatura del resolver

Note

Ora supportiamo principalmente il runtime APPSYNC_JS e la relativa documentazione.

[Prendi in considerazione l'utilizzo del runtime APPSYNC_JS e delle relative guide qui.](#)

AWS AppSync definisce un insieme di variabili e funzioni per lavorare con i modelli di mappatura dei resolver. Ciò semplifica le operazioni logiche sui dati con GraphQL. Questo documento descrive queste funzioni e offre alcuni esempi per l'uso dei modelli.

Accesso a **\$context**

La variabile `$context` è una mappa contenente tutte le informazioni contestuali per la chiamata del resolver. Ha la struttura seguente:

```
{
  "arguments" : { ... },
  "source" : { ... },
```

```
"result" : { ... },
"identity" : { ... },
"request" : { ... },
"info": { ... }
}
```

Note

Se state cercando di accedere a una dictionary/map voce (ad esempio una voce in `context`) utilizzando la relativa chiave per recuperare il valore, il Velocity Template Language (VTL) consente di utilizzare direttamente la notazione `<dictionary-element>.<key-name>`. Tuttavia, questo potrebbe non funzionare per tutti i casi, ad esempio quando i nomi di chiavi dispongono di caratteri speciali (per esempio, un segno di sottolineatura `_`). Ti consigliamo di utilizzare sempre una notazione `<dictionary-element>.get("<key-name>")`.

Ogni campo nella mappa `$context` è definito nel modo seguente:

\$context campi

arguments

Una mappa contenente tutti gli argomenti GraphQL per questo campo.

identity

Un oggetto contenente le informazioni sul chiamante. Vedi [Identità](#) per ulteriori informazioni sulla struttura di questo campo.

source

Una mappa contenente la risoluzione del campo padre.

stash

Lo stash è una mappa disponibile all'interno di ogni modello di mappatura di funzione o resolver. La sua istanza viene attivata dall'esecuzione del resolver. Ciò significa che è possibile trasferire arbitrariamente i dati tra i modelli di mappatura della richiesta e della risposta e tra le funzioni di un resolver di pipeline. Lo stash ammette gli stessi metodi previsti dalla struttura di dati di una [mappa Java](#).

result

Un container per i risultati di questo resolver. Questo campo è disponibile solo per i modelli di mappatura delle risposte.

Ad esempio, se stai risolvendo il `author` campo della seguente query:

```
query {  
  getPost(id: 1234) {  
    postId  
    title  
    content  
    author {  
      id  
      name  
    }  
  }  
}
```

La variabile `$context` completa disponibile durante l'elaborazione di un modello di mappatura della risposta potrebbe essere:

```
{  
  "arguments" : {  
    id: "1234"  
  },  
  "source": {},  
  "result" : {  
    "postId": "1234",  
    "title": "Some title",  
    "content": "Some content",  
    "author": {  
      "id": "5678",  
      "name": "Author Name"  
    }  
  },  
  "identity" : {  
    "sourceIp" : ["x.x.x.x"],  
    "userArn" : "arn:aws:iam::123456789012:user/appsync",  
    "accountId" : "666666666666",  
    "user" : "AIDAAAAAAAAAAAAAAAAAAAA"  
  }  
}
```

```
}
```

prev.result

Il risultato di qualsiasi operazione precedente sia stata eseguita in un resolver di pipeline.

Se l'operazione precedente era il modello Before mapping del risolutore di pipeline, allora `$ctx.prev.result` rappresenta l'output della valutazione del modello e viene reso disponibile per la prima funzione nella pipeline.

Se l'operazione precedente corrisponde alla prima funzione, `$ctx.prev.result` è l'output della prima funzione, disponibile per la seconda funzione della pipeline.

Se l'operazione precedente era l'ultima funzione, `$ctx.prev.result` rappresenta l'output dell'ultima funzione e viene resa disponibile al modello After mapping del risolutore di pipeline.

info

Un oggetto contenente le informazioni sulla richiesta GraphQL. Per la struttura di questo campo, consulta [Info](#).

Identità

La sezione `identity` contiene le informazioni sul chiamante. La forma di questa sezione dipende dal tipo di autorizzazione dell'API. AWS AppSync

Per ulteriori informazioni sulle opzioni di AWS AppSync sicurezza, consulta [Autorizzazione e autenticazione](#).

Autorizzazione **API_KEY**

Il `identity` campo non è compilato.

Autorizzazione **AWS_LAMBDA**

`identity` Contiene la `resolverContext` chiave, contenente lo stesso `resolverContext` contenuto restituito dalla funzione Lambda che autorizza la richiesta.

Autorizzazione **AWS_IAM**

`identity` Ha la seguente forma:

```
{
```

```
"accountId" : "string",
"cognitoIdentityPoolId" : "string",
"cognitoIdentityId" : "string",
"sourceIp" : ["string"],
"username" : "string", // IAM user principal
"userArn" : "string",
"cognitoIdentityAuthType" : "string", // authenticated/unauthenticated based on
the identity type
"cognitoIdentityAuthProvider" : "string" // the auth provider that was used to
obtain the credentials
}
```

Autorizzazione **AMAZON_COGNITO_USER_POOLS**

identityHa la forma seguente:

```
{
  "sub" : "uuid",
  "issuer" : "string",
  "username" : "string"
  "claims" : { ... },
  "sourceIp" : ["x.x.x.x"],
  "defaultAuthStrategy" : "string"
}
```

Ogni campo è definito nel modo seguente:

accountId

L'ID dell' AWS account del chiamante.

claims

Le attestazioni dell'utente.

cognitoIdentityAuthType

Autenticato o non autenticato in base al tipo di identità.

cognitoIdentityAuthProvider

Un elenco separato da virgole di informazioni sul provider di identità esterno utilizzato per ottenere le credenziali utilizzate per firmare la richiesta.

cognitoIdentityId

L'ID identificativo Amazon Cognito del chiamante.

cognitoIdentityPoolId

L'ID del pool di identità di Amazon Cognito associato al chiamante.

defaultAuthStrategy

La strategia di autorizzazione predefinita per questo chiamante (ALLOW o DENY).

issuer

L'emittente del token.

sourceIp

L'indirizzo IP di origine del chiamante che riceve. AWS AppSync Se la richiesta non include l'`x-forwarded-for` intestazione, il valore IP di origine contiene solo un singolo indirizzo IP della connessione TCP. Se la richiesta include un'intestazione `x-forwarded-for`, l'IP di origine sarà un elenco di indirizzi IP dell'intestazione `x-forwarded-for` oltre all'indirizzo IP proveniente dalla connessione TCP.

sub

L'UUID dell'utente autenticato.

user

L'utente IAM.

userArn

L'Amazon Resource Name (ARN) dell'utente IAM.

username

Il nome dell'utente autenticato. In caso di autorizzazione `AMAZON_COGNITO_USER_POOLS`, il valore del nome utente è il valore dell'attributo `cognito:username`. In caso di `AWS_IAM` autorizzazione, il valore del nome utente è il valore del principale AWS utente. Se utilizzi l'autorizzazione IAM con credenziali fornite dai pool di identità di Amazon Cognito, ti consigliamo di utilizzare `cognitoIdentityId`

Intestazioni delle richieste di accesso

AWS AppSync supporta il passaggio di intestazioni personalizzate dai client e l'accesso ad esse nei resolver GraphQL utilizzando `$context.request.headers`. È quindi possibile utilizzare i valori dell'intestazione per azioni come l'inserimento di dati in una fonte di dati o i controlli di autorizzazione. È possibile utilizzare intestazioni di richiesta singole o multiple utilizzando `$curl` una chiave API dalla riga di comando, come illustrato negli esempi seguenti:

Esempio di intestazione singola

Supponi di impostare un'intestazione custom con il valore `nadia` come segue:

```
curl -XPOST -H "Content-Type:application/graphql" -H "custom:nadia" -H "x-api-key:<API-KEY-VALUE>" -d '{"query":"mutation { createEvent(name: \"demo\", when: \"Next Friday!\", where: \"Here!\") {id name when where description}}}"' https://<ENDPOINT>/graphql
```

Potrai quindi accedervi con `$context.request.headers.custom`. Ad esempio, potrebbe trovarsi nel seguente VTL per DynamoDB:

```
"custom": $util.dynamodb.toDynamoDBJson($context.request.headers.custom)
```

Esempio di intestazione multipla

Puoi anche passare intestazioni multiple in una singola richiesta e accedervi nel modello di mappatura del resolver. Ad esempio, se l'intestazione è impostata con due valori:

```
curl -XPOST -H "Content-Type:application/graphql" -H "custom:bailey" -H "custom:nadia" -H "x-api-key:<API-KEY-VALUE>" -d '{"query":"mutation { createEvent(name: \"demo\", when: \"Next Friday!\", where: \"Here!\") {id name when where description}}}"' https://<ENDPOINT>/graphql
```

Potrai quindi accedervi come matrice, ad esempio `$context.request.headers.custom[1]`.

Note

AWS AppSync non espone l'intestazione del cookie in `$context.request.headers`

Accedi al nome di dominio personalizzato della richiesta

AWS AppSync supporta la configurazione di un dominio personalizzato che puoi utilizzare per accedere ai tuoi endpoint GraphQL e in tempo reale per il tuo. APIs Quando si effettua una richiesta con un nome di dominio personalizzato, è possibile ottenere il nome di dominio utilizzando.

`$context.request.domainName`

Quando si utilizza il nome di dominio endpoint GraphQL predefinito, il valore è. `null`

Informazioni

La sezione `info` contiene informazioni sulla richiesta GraphQL. Questa sezione ha il seguente formato:

```
{
  "fieldName": "string",
  "parentTypeName": "string",
  "variables": { ... },
  "selectionSetList": ["string"],
  "selectionSetGraphQL": "string"
}
```

Ogni campo è definito nel modo seguente:

fieldName

Il nome del campo attualmente in corso di risoluzione.

parentTypeName

Il nome del tipo padre per il campo attualmente in corso di risoluzione.

variables

Una mappa contenente tutte le variabili che vengono passate nella richiesta GraphQL.

selectionSetList

Rappresentazione elenco dei campi nel set di selezione GraphQL. I campi con alias sono referenziati solo dal nome dell'alias, non dal nome del campo. L'esempio seguente mostra questa indicazione in dettaglio.

selectionSetGraphQL

Rappresentazione stringa del set di selezione, formattata come SDL (Schema Definition Language) GraphQL. Sebbene i frammenti non vengano uniti nel set di selezione, i frammenti in linea vengono conservati, come illustrato nell'esempio seguente.

Note

Quando si utilizza `$utils.toJson() oncontext.info`, i valori che `selectionSetGraphQL` e `selectionSetList` restituiscono non vengono serializzati per impostazione predefinita.

Ad esempio, se stai risolvendo il campo `getPost` della query seguente:

```
query {
  getPost(id: $postId) {
    postId
    title
    secondTitle: title
    content
    author(id: $authorId) {
      authorId
      name
    }
    secondAuthor(id: "789") {
      authorId
    }
    ... on Post {
      inlineFrag: comments: {
        id
      }
    }
    ... postFrag
  }
}

fragment postFrag on Post {
  postFrag: comments: {
    id
  }
}
```

```
}
```

La variabile `$context.info` completa disponibile durante l'elaborazione di un modello di mappatura potrebbe essere:

```
{
  "fieldName": "getPost",
  "parentTypeName": "Query",
  "variables": {
    "postId": "123",
    "authorId": "456"
  },
  "selectionSetList": [
    "postId",
    "title",
    "secondTitle",
    "content",
    "author",
    "author/authorId",
    "author/name",
    "secondAuthor",
    "secondAuthor/authorId",
    "inlineFragComments",
    "inlineFragComments/id",
    "postFragComments",
    "postFragComments/id"
  ],
  "selectionSetGraphQL": "{\n  getPost(id: $postId) {\n    postId\n    title\n    secondTitle: title\n    content\n    author(id: $authorId) {\n      authorId\n      name\n    }\n    secondAuthor(id: \"789\") {\n      authorId\n    }\n    ... on Post\n    {\n      inlineFrag: comments {\n        id\n      }\n    }\n    ... postFrag\n  }\n}"
}
```

`selectionSetList` espone solo i campi che appartengono al tipo corrente. Se il tipo corrente è un'interfaccia o un'unione, vengono esposti solo i campi selezionati che appartengono all'interfaccia. Ad esempio, dato lo schema seguente:

```
type Query {
  node(id: ID!): Node
}

interface Node {
```

```
    id: ID
  }

  type Post implements Node {
    id: ID
    title: String
    author: String
  }

  type Blog implements Node {
    id: ID
    title: String
    category: String
  }
```

E la seguente domanda:

```
query {
  node(id: "post1") {
    id
    ... on Post {
      title
    }

    ... on Blog {
      title
    }
  }
}
```

Quando si chiama `$ctx.info.selectionSetList` alla risoluzione del `Query.node` campo, `id` viene esposto solo:

```
"selectionSetList": [
  "id"
]
```

Neutralizzazione degli input

Le applicazioni devono neutralizzare gli input non attendibili per impedire a qualsiasi parte esterna di utilizzare un'applicazione al di fuori dell'uso previsto. Poiché

`$context` contiene gli input degli utenti in proprietà come `$context.arguments`,,
`$context.request.headers`,`$context.identity`,`$context.result`,`$context.info.variables`.
È necessario prestare attenzione a ripulirne i valori nei modelli di mappatura.

Poiché i modelli di mapping sono rappresentati in JSON, la neutralizzazione degli input assume la forma di caratteri riservati JSON di escape delle stringhe che rappresentano gli input dell'utente. È consigliabile utilizzare l'utilità `$util.toJson()` per applicare i caratteri riservati JSON di escape di valori di stringa sensibili quando vengono inseriti in un modello di mapping.

Ad esempio, nel seguente modello di mappatura delle richieste Lambda, poiché abbiamo avuto accesso a una stringa di input del cliente non sicura (`$context.arguments.id`), l'abbiamo inserita `$util.toJson()` per evitare che i caratteri JSON senza escape violassero il modello JSON.

```
{
  "version": "2017-02-28",
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "postId": $util.toJson($context.arguments.id)
  }
}
```

A differenza del modello di mappatura riportato di seguito, in cui inseriamo direttamente senza sanificazione. `$context.arguments.id` Questo non funziona per le stringhe che contengono virgolette senza scampo o altri caratteri riservati JSON e può lasciare il modello esposto a errori.

```
## DO NOT DO THIS
{
  "version": "2017-02-28",
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "postId": "$context.arguments.id" ## Unsafe! Do not insert $context string
    values without escaping JSON characters.
  }
}
```

AWS AppSync riferimento all'utilità del modello di mappatura del resolver

Note

Ora supportiamo principalmente il runtime APPSYNC_JS e la relativa documentazione.

[Prendi in considerazione l'utilizzo del runtime APPSYNC_JS e delle relative guide qui.](#)

AWS AppSync definisce un set di utilità che è possibile utilizzare all'interno di un resolver GraphQL per semplificare le interazioni con le fonti di dati. Alcune di queste utilità sono destinate all'uso generale con qualsiasi fonte di dati, ad esempio la generazione o i timestamp. IDs Altre sono specifiche per un tipo di origine dati. Sono disponibili le seguenti utilità:

- [Utility helper in \\$util](#) - La variabile \$util contiene metodi di utilità generali per facilitare l'utilizzo dei dati. Se non diversamente specificato, tutte le utilità usano il set di caratteri UTF-8.
- [AppSync direttive: AppSync espone le](#) direttive per facilitare la produttività degli sviluppatori durante la scrittura in VTL.
- [Time helpers in \\$util.time - La variabile \\$util.time](#) contiene metodi datetime per aiutare a generare timestamp, convertire tra formati datetime e analizzare stringhe datetime. Si basa sulla sintassi per i formati datetime, a cui è possibile fare riferimento per ulteriore documentazione. [DateTimeFormatter](#)
- [List helpers in \\$util.list - \\$util.list](#) contiene metodi per facilitare le operazioni List più comuni, come la rimozione o il mantenimento di elementi da un elenco per filtrare i casi d'uso.
- [Map helpers in \\$util.map - \\$util.map](#) contiene metodi per facilitare le operazioni più comuni sulle mappe, come la rimozione o il mantenimento di elementi da una mappa per filtrare i casi d'uso.
- [Helper DynamoDB in \\$util.dynamodb - \\$util.dynamodb contiene metodi di supporto che semplificano la scrittura e la lettura dei dati su Amazon DynamoDB, come la mappatura e la formattazione automatiche dei tipi.](#)
- [Gli helper di Amazon RDS in \\$util.rds - \\$util.rds](#) contengono metodi di supporto che formattano le operazioni RDS eliminando i dati estranei negli output dei risultati.
- [Helper HTTP in \\$util.http - L'utilità \\$util.http fornisce metodi di supporto che puoi usare per gestire i parametri di richiesta HTTP](#) e aggiungere intestazioni di risposta.
- [Helper XML in \\$util.xml - \\$util.xml](#) contiene metodi di supporto che possono semplificare la traduzione delle risposte XML in JSON o in un dizionario.

- [Aiutanti di trasformazione in \\$util.transform - \\$util.transform](#) contiene metodi di supporto che semplificano l'esecuzione di operazioni complesse su fonti di dati, come le operazioni di filtro DynamoDB.
- [Aiutanti matematici in \\$util.math - \\$util.math](#) contiene metodi per facilitare le operazioni matematiche più comuni.
- [String helpers in \\$util.str - \\$util.str](#) contiene metodi per facilitare le operazioni più comuni sulle stringhe.
- [Extensions - \\$extensions](#) contiene una serie di metodi per eseguire azioni aggiuntive all'interno dei resolver.

Utility helper in \$util

Note

Ora supportiamo principalmente il runtime APPSYNC_JS e la relativa documentazione. [Prendi in considerazione l'utilizzo del runtime APPSYNC_JS e delle relative guide qui.](#)

La `$util` variabile contiene metodi di utilità generali per aiutarti a lavorare con i dati. Se non diversamente specificato, tutte le utilità usano il set di caratteri UTF-8.

Utilità di analisi JSON

Elenco degli strumenti di analisi JSON

`$util.parseJson(String) : Object`

Da una stringa JSON restituisce una rappresentazione oggetto del risultato.

`$util.toJson(Object) : String`

Da un oggetto restituisce una rappresentazione JSON "a stringhe" di tale oggetto.

Utilità di codifica

Elenco degli strumenti di codifica

\$util.urlEncode(String) : String

Restituisce la stringa di input come stringa codificata `application/x-www-form-urlencoded`.

\$util.urlDecode(String) : String

Decodifica una stringa codificata `application/x-www-form-urlencoded` nella relativa forma non codificata.

\$util.base64Encode(byte[]) : String

Codifica l'input in una stringa con codifica base64.

\$util.base64Decode(String) : byte[]

Decodifica i dati da una stringa con codifica base64.

Utilità per la generazione di ID

Elenco di utilità per la generazione di ID

\$util.autoId() : String

Restituisce un valore UUID generato casualmente a 128 bit.

\$util.autoUlid() : String

Restituisce un ULID (Universally Unique Lexicographically Sortable Identifier) generato casualmente a 128 bit.

\$util.autoKsuid() : String

Restituisce un KSUID (K-Sortable Unique Identifier) base62 generato casualmente a 128 bit codificato come String con una lunghezza di 27.

Utili di errore

Elenco delle utilità di errore

\$util.error(String)

Genera un errore personalizzato. Utilizzalo nei modelli di mappatura delle richieste o delle risposte per rilevare un errore nella richiesta o nel risultato della chiamata.

\$util.error(String, String)

Genera un errore personalizzato. Usalo nei modelli di mappatura delle richieste o delle risposte per rilevare un errore nella richiesta o nel risultato della chiamata. Puoi anche specificare un `errorType`

\$util.error(String, String, Object)

Genera un errore personalizzato. Utilizzalo nei modelli di mappatura delle richieste o delle risposte per rilevare un errore nella richiesta o nel risultato della chiamata. Puoi anche specificare un campo `errorType` e un `data`. Il valore di `data` verrà aggiunto al blocco `error` corrispondente all'interno di `errors` nella risposta di GraphQL.

Note

`data` verrà filtrato in base al set di selezione dell'interrogazione.

\$util.error(String, String, Object, Object)

Genera un errore personalizzato. Può essere usato nei modelli di mappatura di richieste o risposte se il modello rileva un errore nella richiesta o nel risultato della chiamata. Inoltre, è possibile specificare un campo `data`, un campo `errorInfo` e un campo `errorInfoNON`. Il valore di `data` verrà aggiunto al blocco `error` corrispondente all'interno di `errors` nella risposta di GraphQL.

Note

`data` verrà filtrato in base al set di selezione della query. Il valore di `errorInfo` verrà aggiunto al blocco `error` corrispondente all'interno di `errors` nella risposta di GraphQL. `errorInfoNON` verrà filtrato in base al set di selezione delle query.

\$util.appendError(String)

Aggiunge un errore personalizzato. Può essere usato nei modelli di mappatura di richieste o risposte se il modello rileva un errore nella richiesta o nel risultato della chiamata. A differenza di `$util.error(String)`, la valutazione del modello non viene interrotta, in modo che i dati possano essere restituiti al chiamante.

\$util.appendError(String, String)

Aggiunge un errore personalizzato. Può essere usato nei modelli di mappatura di richieste o risposte se il modello rileva un errore nella richiesta o nel risultato della chiamata. È inoltre possibile specificare un campo `errorType`. A differenza di `$util.error(String, String)`, la valutazione del modello non viene interrotta, in modo che i dati possano essere restituiti al chiamante.

\$util.appendError(String, String, Object)

Aggiunge un errore personalizzato. Può essere usato nei modelli di mappatura di richieste o risposte se il modello rileva un errore nella richiesta o nel risultato della chiamata. È inoltre possibile specificare un campo `errorType` e un campo `data`. A differenza di `$util.error(String, String, Object)`, la valutazione del modello non viene interrotta, in modo che i dati possano essere restituiti al chiamante. Il valore di `data` verrà aggiunto al blocco `error` corrispondente all'interno di `errors` nella risposta di GraphQL.

Note

`data` verrà filtrato in base al set di selezione delle query.

\$util.appendError(String, String, Object, Object)

Aggiunge un errore personalizzato. Può essere usato nei modelli di mappatura di richieste o risposte se il modello rileva un errore nella richiesta o nel risultato della chiamata. Inoltre, è possibile specificare un campo `errorType`, un campo `data`, un campo `errorInfo` e un campo `errorPath`. A differenza di `$util.error(String, String, Object, Object)`, la valutazione del modello non viene interrotta, in modo che i dati possano essere restituiti al chiamante. Il valore di `data` verrà aggiunto al blocco `error` corrispondente all'interno di `errors` nella risposta di GraphQL.

 Note

dataverrà filtrato in base al set di selezione della query. Il valore di `errorInfo` verrà aggiunto al blocco `error` corrispondente all'interno di `errors` nella risposta di GraphQL. `errorInfoNON` verrà filtrato in base al set di selezione delle query.

Utilità per la convalida delle condizioni

Elenco degli strumenti di convalida delle condizioni

`$util.validate(Boolean, String) : void`

Se la condizione è falsa, lancia un messaggio `CustomTemplateException` con il messaggio specificato.

`$util.validate(Boolean, String, String) : void`

Se la condizione è falsa, genera un messaggio `CustomTemplateException` con il messaggio e il tipo di errore specificati.

`$util.validate(Boolean, String, String, Object) : void`

Se la condizione è falsa, genera un messaggio `CustomTemplateException` con il messaggio e il tipo di errore specificati, oltre ai dati da restituire nella risposta.

Utilità comportamentali nulle

Elenco di utilità con comportamento nullo

`$util.isNull(Object) : Boolean`

Restituisce `true` se l'oggetto fornito è `null`.

`$util.isNullOrEmpty(String) : Boolean`

Restituisce `true` se i dati forniti sono `null` o una stringa vuota. In caso contrario, restituisce `false`.

`$util.isNullOrBlank(String) : Boolean`

Restituisce `true` se i dati forniti sono `null` o una stringa vuota. In caso contrario, restituisce `false`.

\$util.defaultIfNull(Object, Object) : Object

Restituisce il primo oggetto se non è null. In caso contrario, restituisce il secondo oggetto come "oggetto predefinito".

\$util.defaultIfNullOrEmpty(String, String) : String

Restituisce la prima stringa se non è null o vuota. In caso contrario, restituisce la seconda stringa come "stringa predefinita".

\$util.defaultIfNullOrBlank(String, String) : String

Restituisce la prima stringa se non è null o vuota. In caso contrario, restituisce la seconda stringa come "stringa predefinita".

Utilità per la corrispondenza dei modelli

Elenco degli strumenti per la corrispondenza dei tipi e dei modelli

\$util.typeOf(Object) : String

Restituisce una stringa che descrive il tipo di oggetto. Le identificazioni di tipi supportate sono: "Null", "Number", "String", "Map", "List", "Boolean". Se un tipo non può essere identificato, il tipo restituito è "Object".

\$util.matches(String, String) : Boolean

Restituisce true se il modello specificato nel primo argomento corrisponde ai dati forniti nel secondo argomento. Il modello deve essere un'espressione regolare, ad esempio `$util.matches("a*b", "aaaaab")`. La funzionalità si basa sulla classe [Pattern](#) che puoi consultare per ottenere altre informazioni.

\$util.authType() : String

Restituisce una stringa che descrive il tipo di autenticazione multipla utilizzato da una richiesta, restituendo «IAM Authorization», «User Pool Authorization», «Open ID Connect Authorization» o «API Key Authorization».

Utilità di convalida degli oggetti

Elenco degli strumenti di convalida degli oggetti

\$util.isString(Object) : Boolean

Restituisce true se l'oggetto è una stringa.

\$util.isNumber(Object) : Boolean

Restituisce true se l'oggetto è un numero.

\$util.isBoolean(Object) : Boolean

Restituisce true se l'oggetto è un valore booleano.

\$util.isList(Object) : Boolean

Restituisce true se l'oggetto è un elenco.

\$util.isMap(Object) : Boolean

Restituisce true se l'oggetto è una mappa.

CloudWatch utilità di registrazione

CloudWatch lista di utilità di registrazione

\$util.log.info(Object) : Void

Registra la rappresentazione String dell'oggetto fornito nel flusso di registro richiesto quando la registrazione a livello di richiesta e di campo è abilitata con CloudWatch log level o su un'API. ALL INFO DEBUG

\$util.log.info(String, Object...) : Void

Registra la rappresentazione String degli oggetti forniti nel flusso di registro richiesto quando la registrazione a livello di richiesta e di campo è CloudWatch abilitata con il livello di log su un'API. ALL Questa utilità sostituirà tutte le variabili indicate da «{ }» nella prima stringa di formato di input con la rappresentazione String degli oggetti forniti nell'ordine.

\$util.log.debug(Object) : Void

Registra la rappresentazione String dell'oggetto fornito nel flusso di registro richiesto quando la registrazione a livello di richiesta e di campo è abilitata con il livello CloudWatch di log o su un'API. ALL DEBUG

\$util.log.debug(String, Object...) : Void

Registra la rappresentazione String degli oggetti forniti nel flusso di registro richiesto quando la registrazione a livello di campo è abilitata con livello di registro o livello CloudWatch di registro su un'API. DEBUG ALL Questa utilità sostituirà tutte le variabili indicate da «{}» nella prima stringa di formato di input con la rappresentazione String degli oggetti forniti nell'ordine.

\$util.log.error(Object) : Void

Registra la rappresentazione in formato String dell'oggetto fornito nel flusso di log richiesto quando la CloudWatch registrazione a livello di campo è abilitata con qualsiasi livello di registro (ALL, INFODEBUG, ecc.) su un'API.

\$util.log.error(String, Object...) : Void

Registra la rappresentazione String degli oggetti forniti nel flusso di registro richiesto quando la registrazione a livello di campo è abilitata con livello di CloudWatch registro o livello di registro su un'API. ERROR ALL Questa utilità sostituirà tutte le variabili indicate da «{}» nella prima stringa di formato di input con la rappresentazione String degli oggetti forniti nell'ordine.

Restituisce il valore di comportamento (utils)

Elenco delle utilità di comportamento del valore restituito

\$util.qr() e \$util.quiet()

Esegue un'istruzione VTL mentre sopprime il valore restituito. Ciò è utile per eseguire metodi senza utilizzare segnaposti temporanei, ad esempio aggiungere elementi a una mappa. Esempio:

```
#set ($myMap = {})  
#set($discard = $myMap.put("id", "first value"))
```

Diventa:

```
#set ($myMap = {})  
$util.qr($myMap.put("id", "first value"))
```

\$util.escapeJavaScript(String) : String

Restituisce la stringa di input come stringa di JavaScript escape.

\$util.urlEncode(String) : String

Restituisce la stringa di input come stringa codificata `application/x-www-form-urlencoded`.

\$util.urlDecode(String) : String

Decodifica una stringa codificata `application/x-www-form-urlencoded` nella relativa forma non codificata.

\$util.base64Encode(byte[]) : String

Codifica l'input in una stringa con codifica base64.

\$util.base64Decode(String) : byte[]

Decodifica i dati da una stringa con codifica base64.

\$util.parseJson(String) : Object

Da una stringa JSON restituisce una rappresentazione oggetto del risultato.

\$util.toJson(Object) : String

Da un oggetto restituisce una rappresentazione JSON "a stringhe" di tale oggetto.

\$util.autoId() : String

Restituisce un valore UUID generato casualmente a 128 bit.

\$util.autoUlid() : String

Restituisce un ULID (Universally Unique Lexicographically Sortable Identifier) generato casualmente a 128 bit.

\$util.autoKsuid() : String

Restituisce un KSUID (K-Sortable Unique Identifier) base62 generato casualmente a 128 bit codificato come String con una lunghezza di 27.

\$util.unauthorized()

Genera Unauthorized per il campo in fase di risoluzione. Utilizzalo nei modelli di mappatura delle richieste o delle risposte per determinare se consentire al chiamante di risolvere il campo.

\$util.error(String)

Genera un errore personalizzato. Utilizzalo nei modelli di mappatura delle richieste o delle risposte per rilevare un errore nella richiesta o nel risultato della chiamata.

\$util.error(String, String)

Genera un errore personalizzato. Usalo nei modelli di mappatura delle richieste o delle risposte per rilevare un errore nella richiesta o nel risultato della chiamata. Puoi anche specificare un. `errorType`

\$util.error(String, String, Object)

Genera un errore personalizzato. Utilizzalo nei modelli di mappatura delle richieste o delle risposte per rilevare un errore nella richiesta o nel risultato della chiamata. Puoi anche specificare un campo `errorType` e un. `data` Il valore di `data` verrà aggiunto al blocco `error` corrispondente all'interno di `errors` nella risposta di GraphQL. Nota: `data` verrà filtrato in base al set di selezioni della query.

\$util.error(String, String, Object, Object)

Genera un errore personalizzato. Può essere usato nei modelli di mappatura di richieste o risposte se il modello rileva un errore nella richiesta o nel risultato della chiamata. È inoltre possibile specificare un campo `errorType`, un campo `data` e un campo `errorInfo`. Il valore di `data` verrà aggiunto al blocco `error` corrispondente all'interno di `errors` nella risposta di GraphQL. Nota: `data` verrà filtrato in base al set di selezioni della query. Il valore di `errorInfo` verrà aggiunto al blocco `error` corrispondente all'interno di `errors` nella risposta di GraphQL. Nota: `errorInfo` NON verrà filtrato in base al set di selezioni della query.

\$util.appendError(String)

Aggiunge un errore personalizzato. Può essere usato nei modelli di mappatura di richieste o risposte se il modello rileva un errore nella richiesta o nel risultato della chiamata. A differenza di `$util.error(String)`, la valutazione del modello non viene interrotta, in modo che i dati possano essere restituiti al chiamante.

\$util.appendError(String, String)

Aggiunge un errore personalizzato. Può essere usato nei modelli di mappatura di richieste o risposte se il modello rileva un errore nella richiesta o nel risultato della chiamata. È inoltre possibile specificare un campo `errorType`. A differenza di `$util.error(String, String)`, la valutazione del modello non viene interrotta, in modo che i dati possano essere restituiti al chiamante.

\$util.appendError(String, String, Object)

Aggiunge un errore personalizzato. Può essere usato nei modelli di mappatura di richieste o risposte se il modello rileva un errore nella richiesta o nel risultato della chiamata. È inoltre possibile specificare un campo `errorType` e un campo `data`. A differenza di `$util.error(String, String, Object)`, la valutazione del modello non viene interrotta, in modo che i dati possano essere restituiti al chiamante. Il valore di `data` verrà aggiunto al blocco `error` corrispondente all'interno di `errors` nella risposta di GraphQL. Nota: `data` verrà filtrato in base al set di selezioni della query.

\$util.appendError(String, String, Object, Object)

Aggiunge un errore personalizzato. Può essere usato nei modelli di mappatura di richieste o risposte se il modello rileva un errore nella richiesta o nel risultato della chiamata. È inoltre possibile specificare un campo `errorType`, un campo `data` e un campo `errorInfo`. A differenza di `$util.error(String, String, Object, Object)`, la valutazione del modello non viene interrotta, in modo che i dati possano essere restituiti al chiamante. Il valore di `data` verrà aggiunto al blocco `error` corrispondente all'interno di `errors` nella risposta di GraphQL. Nota: `data` verrà filtrato in base al set di selezioni della query. Il valore di `errorInfo` verrà aggiunto al blocco `error` corrispondente all'interno di `errors` nella risposta di GraphQL. Nota: `errorInfo` NON verrà filtrato in base al set di selezioni della query.

\$util.validate(Boolean, String) : void

Se la condizione è falsa, lancia un messaggio `CustomTemplateException` con il messaggio specificato.

\$util.validate(Boolean, String, String) : void

Se la condizione è falsa, genera un messaggio `CustomTemplateException` con il messaggio e il tipo di errore specificati.

\$util.validate(Boolean, String, String, Object) : void

Se la condizione è falsa, genera un messaggio `CustomTemplateException` con il messaggio e il tipo di errore specificati, oltre ai dati da restituire nella risposta.

\$util.isNull(Object) : Boolean

Restituisce `true` se l'oggetto fornito è `null`.

\$util.isNullOrEmpty(String) : Boolean

Restituisce true se i dati forniti sono null o una stringa vuota. In caso contrario, restituisce false.

\$util.isNullOrBlank(String) : Boolean

Restituisce true se i dati forniti sono null o una stringa vuota. In caso contrario, restituisce false.

\$util.defaultIfNull(Object, Object) : Object

Restituisce il primo oggetto se non è null. In caso contrario, restituisce il secondo oggetto come "oggetto predefinito".

\$util.defaultIfNullOrEmpty(String, String) : String

Restituisce la prima stringa se non è null o vuota. In caso contrario, restituisce la seconda stringa come "stringa predefinita".

\$util.defaultIfNullOrBlank(String, String) : String

Restituisce la prima stringa se non è null o vuota. In caso contrario, restituisce la seconda stringa come "stringa predefinita".

\$util.isString(Object) : Boolean

Restituisce true se l'oggetto è una stringa.

\$util.isNumber(Object) : Boolean

Restituisce true se l'oggetto è un numero.

\$util.isBoolean(Object) : Boolean

Restituisce true se l'oggetto è un valore booleano.

\$util.isList(Object) : Boolean

Restituisce true se l'oggetto è un elenco.

\$util.isMap(Object) : Boolean

Restituisce true se l'oggetto è una mappa.

\$util.typeOf(Object) : String

Restituisce una stringa che descrive il tipo di oggetto. Le identificazioni di tipi supportate sono: "Null", "Number", "String", "Map", "List", "Boolean". Se un tipo non può essere identificato, il tipo restituito è "Object".

\$util.matches(String, String) : Boolean

Restituisce true se il modello specificato nel primo argomento corrisponde ai dati forniti nel secondo argomento. Il modello deve essere un'espressione regolare, ad esempio `$util.matches("a*b", "aaaaab")`. La funzionalità si basa sulla classe [Pattern](#) che puoi consultare per ottenere altre informazioni.

\$util.authType() : String

Restituisce una stringa che descrive il tipo di autenticazione multipla utilizzato da una richiesta, restituendo «IAM Authorization», «User Pool Authorization», «Open ID Connect Authorization» o «API Key Authorization».

\$util.log.info(Object) : Void

Registra la rappresentazione in formato String dell'oggetto fornito nel flusso di registro richiesto quando la registrazione a livello di richiesta e di campo è abilitata con il livello di log su CloudWatch un'API. ALL

\$util.log.info(String, Object...) : Void

Registra la rappresentazione String degli oggetti forniti nel flusso di registro richiesto quando la registrazione a livello di richiesta e di campo è CloudWatch abilitata con il livello di log su un'API. ALL Questa utilità sostituirà tutte le variabili indicate da «{ }» nella prima stringa di formato di input con la rappresentazione String degli oggetti forniti nell'ordine.

\$util.log.error(Object) : Void

Registra la rappresentazione String dell'oggetto fornito nel flusso di log richiesto quando la CloudWatch registrazione a livello di campo è abilitata con livello di registro ERROR o livello di registro su un'API. ALL

\$util.log.error(String, Object...) : Void

Registra la rappresentazione String degli oggetti forniti nel flusso di registro richiesto quando la registrazione a livello di campo è abilitata con livello di CloudWatch registro o livello di registro su un'API. ERROR ALL Questa utilità sostituirà tutte le variabili indicate da «{ }» nella prima stringa di formato di input con la rappresentazione String degli oggetti forniti nell'ordine.

\$util.escapeJavaScript(String) : String

Restituisce la stringa di input come stringa JavaScript di escape.

Autorizzazione Resolver

Elenco di autorizzazioni del resolver

`$util.unauthorized()`

Genera Unauthorized per il campo in fase di risoluzione. Utilizzalo nei modelli di mappatura delle richieste o delle risposte per determinare se consentire al chiamante di risolvere il campo.

AWS AppSync direttive

Note

Ora supportiamo principalmente il runtime APPSYNC_JS e la relativa documentazione.
[Prendi in considerazione l'utilizzo del runtime APPSYNC_JS e delle relative guide qui.](#)

AWS AppSync espone direttive per facilitare la produttività degli sviluppatori durante la scrittura in VTL.

Utilità direttive

#return(Object)

`#return(Object)` Consente di tornare prematuramente da qualsiasi modello di mappatura. `#return(Object)` è analoga alla parola chiave `return` nei linguaggi di programmazione, in quanto restituirà dal blocco logico con ambito più vicino. L'utilizzo `#return(Object)` all'interno di un modello di mappatura del resolver verrà restituito dal resolver. Inoltre, l'utilizzo `#return(Object)` di un modello di mappatura delle funzioni ritornerà dalla funzione e proseguirà l'esecuzione verso la funzione successiva nella pipeline o il modello di mappatura delle risposte del resolver.

#return

La `#return` direttiva mostra gli stessi comportamenti di, ma verrà invece restituita `#return(Object).null`

Aiutanti temporali in \$util.time

Note

Ora supportiamo principalmente il runtime APPSYNC_JS e la relativa documentazione. [Prendi in considerazione l'utilizzo del runtime APPSYNC_JS e delle relative guide qui.](#)

La variabile `$util.time` contiene metodi datetime che consentono di generare timestamp, convertire i diversi formati datetime e analizzare le stringhe datetime. La sintassi per i formati datetime si basa sulla [DateTimeFormatter](#) quale è possibile fare riferimento per ulteriore documentazione. Di seguito forniamo alcuni esempi, oltre a un elenco di metodi e descrizioni disponibili.

Utilità temporali

Elenco delle utilità temporali

`$util.time.nowISO8601()` : String

[Restituisce una rappresentazione String di UTC in ISO8601 formato.](#)

`$util.time.nowEpochSeconds()` : long

Restituisce il numero di secondi dall'epoca (Unix epoch) 1970-01-01T00:00:00Z a ora.

`$util.time.nowEpochMilliseconds()` : long

Restituisce il numero di millisecondi dall'epoca (Unix epoch) 1970-01-01T00:00:00Z a ora.

`$util.time.nowFormatted(String)` : String

Restituisce una stringa del timestamp corrente in UTC utilizzando il formato specificato da un tipo di input stringa.

`$util.time.nowFormatted(String, String)` : String

Restituisce una stringa del timestamp corrente per un fuso orario utilizzando il formato e il fuso orario specificati da tipi di input stringa.

`$util.time.parseFormattedToEpochMilliseconds(String, String)` : Long

Analizza un timestamp passato come String insieme a un formato contenente un fuso orario, quindi restituisce il timestamp in millisecondi dall'epoca.

\$util.time.parseFormattedToEpochMilliseconds(String, String, String) : Long

Analizza un timestamp passato come String insieme a un formato e un fuso orario, quindi restituisce il timestamp in millisecondi dall'epoca.

\$util.time.parseISO8601ToEpochMilliseconds(String) : Long

Analizza un ISO8601 timestamp passato come String, quindi restituisce il timestamp in millisecondi trascorsi da epoch.

\$util.time.epochMillisecondsToSeconds(long) : long

Converte un timestamp in formato epoca (Unix epoch) espresso in millisecondi in un timestamp in formato epoca (Unix epoch) espresso in secondi.

\$util.time.epochMillisecondsToISO8601(long) : String

Converte il timestamp di un'epoca in millisecondi in un timestamp. ISO8601

\$util.time.epochMillisecondsToFormatted(long, String) : String

Converte il timestamp di un'epoca in millisecondi, passato così a lungo, in un timestamp formattato secondo il formato fornito in UTC.

\$util.time.epochMillisecondsToFormatted(long, String, String) : String

Converte il timestamp di un'epoca in millisecondi, passato come long, in un timestamp formattato secondo il formato fornito nel fuso orario fornito.

Esempi di funzioni autonome

```
$util.time.nowISO8601() : 
2018-02-06T19:01:35.749Z
$util.time.nowEpochSeconds() : 1517943695
$util.time.nowEpochMilliseconds() : 1517943695750
$util.time.nowFormatted("yyyy-MM-dd HH:mm:ssZ") : 2018-02-06
19:01:35+0000
$util.time.nowFormatted("yyyy-MM-dd HH:mm:ssZ", "+08:00") : 2018-02-07
03:01:35+0800
$util.time.nowFormatted("yyyy-MM-dd HH:mm:ssZ", "Australia/Perth") : 2018-02-07
03:01:35+0800
```

Esempi di conversione

```
#set( $nowEpochMillis = 1517943695758 )
```

```
$util.time.epochMillisecondsToSeconds($nowEpochMillis)
    : 1517943695
$util.time.epochMillisecondsToISO8601($nowEpochMillis)
    : 2018-02-06T19:01:35.758Z
$util.time.epochMillisecondsToFormatted($nowEpochMillis, "yyyy-MM-dd HH:mm:ssZ")
    : 2018-02-06 19:01:35+0000
$util.time.epochMillisecondsToFormatted($nowEpochMillis, "yyyy-MM-dd HH:mm:ssZ",
    "+08:00") : 2018-02-07 03:01:35+0800
```

Esempi di analisi

```
$util.time.parseISO8601ToEpochMilliseconds("2018-02-01T17:21:05.180+08:00")
    : 1517476865180
$util.time.parseFormattedToEpochMilliseconds("2018-02-02 01:19:22+0800", "yyyy-MM-dd
    HH:mm:ssZ") : 1517505562000
$util.time.parseFormattedToEpochMilliseconds("2018-02-02 01:19:22", "yyyy-MM-dd
    HH:mm:ss", "+08:00") : 1517505562000
```

Utilizzo con scalari AWS AppSync definiti

I seguenti formati sono compatibili con `AWSDate`, `AWSDatetime` e `AWSTime`.

```
$util.time.nowFormatted("yyyy-MM-dd[XXX]", "-07:00:30") :
    2018-07-11-07:00
$util.time.nowFormatted("yyyy-MM-dd'T'HH:mm:ss[XXXXXX]", "-07:00:30") :
    2018-07-11T15:14:15-07:00:30
```

Elenca gli aiutanti in `$util.list`

Note

Ora supportiamo principalmente il runtime `APPSYNC_JS` e la relativa documentazione.
[Prendi in considerazione l'utilizzo del runtime APPSYNC_JS e delle relative guide qui.](#)

`$util.list` contiene metodi per facilitare le operazioni più comuni di `List`, come la rimozione o il mantenimento di elementi da un elenco per filtrare i casi d'uso.

Elenca gli strumenti

\$util.list.copyAndRetainAll(List, List) : List

Crea una copia superficiale dell'elenco fornito nel primo argomento conservando solo gli elementi specificati nel secondo argomento, se presenti. Tutti gli altri elementi verranno rimossi dalla copia.

\$util.list.copyAndRemoveAll(List, List) : List

Crea una copia superficiale dell'elenco fornito nel primo argomento rimuovendo tutti gli elementi in cui l'elemento è specificato nel secondo argomento, se presenti. Tutti gli altri elementi verranno mantenuti nella copia.

\$util.list.sortList(List, Boolean, String) : List

Ordina un elenco di oggetti, fornito nel primo argomento. Se il secondo argomento è vero, l'elenco viene ordinato in modo decrescente; se il secondo argomento è falso, l'elenco viene ordinato in modo crescente. Il terzo argomento è il nome della stringa della proprietà utilizzata per ordinare un elenco di oggetti personalizzati. Se si tratta di un elenco di sole stringhe, numeri interi, float o doppi, il terzo argomento può essere qualsiasi stringa casuale. Se tutti gli oggetti non appartengono alla stessa classe, viene restituito l'elenco originale. Sono supportati solo gli elenchi contenenti un massimo di 1000 oggetti. Di seguito è riportato un esempio di utilizzo di questa utilità:

```
INPUT:      $util.list.sortList([{"description":"youngest", "age":5},
{"description":"middle", "age":45}, {"description":"oldest", "age":85}], false,
"description")
OUTPUT:     [{"description":"middle", "age":45}, {"description":"oldest",
"age":85}, {"description":"youngest", "age":5}]
```

Aiutanti di mappe in \$util.map

Note

Ora supportiamo principalmente il runtime APPSYNC_JS e la relativa documentazione. [Prendi in considerazione l'utilizzo del runtime APPSYNC_JS e delle relative guide qui.](#)

`$util.map` contiene metodi per facilitare le operazioni più comuni della mappa, come la rimozione o il mantenimento di elementi da una mappa per filtrare i casi d'uso.

Utilità della mappa

`$util.map.copyAndRetainAllKeys(Map, List) : Map`

Crea una copia superficiale della prima mappa conservando solo le chiavi specificate nell'elenco, se presenti. Tutte le altre chiavi verranno rimosse dalla copia.

`$util.map.copyAndRemoveAllKeys(Map, List) : Map`

Crea una copia superficiale della prima mappa rimuovendo tutte le voci in cui la chiave è specificata nell'elenco, se presenti. Tutte le altre chiavi verranno mantenute nella copia.

Helper DynamoDB in `$util.dynamodb`

Note

Ora supportiamo principalmente il runtime APPSYNC_JS e la relativa documentazione. [Prendi in considerazione l'utilizzo del runtime APPSYNC_JS e delle relative guide qui.](#)

`$util.dynamodb` contiene metodi di supporto che semplificano la scrittura e la lettura dei dati su Amazon DynamoDB, come la mappatura e la formattazione automatiche dei tipi. Questi metodi sono progettati per mappare automaticamente i tipi e gli elenchi primitivi nel formato di input DynamoDB corretto, che è uno dei formati. `Map { "TYPE" : VALUE }`

Ad esempio, in precedenza, un modello di mappatura delle richieste per creare un nuovo elemento in DynamoDB avrebbe potuto avere il seguente aspetto:

```
{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key": {
    "id" : { "S" : "$util.autoId()" }
  },
  "attributeValues" : {
    "title" : { "S" : $util.toJson($ctx.args.title) },
    "author" : { "S" : $util.toJson($ctx.args.author) },
  }
}
```

```

    "version" : { "N", $util.toJson($ctx.args.version) }
  }
}

```

Per aggiungere campi all'oggetto sarebbe stato necessario aggiornare la query GraphQL nello schema, nonché il modello di mappatura della richiesta. Tuttavia, ora possiamo ristrutturare il nostro modello di mappatura delle richieste in modo che raccolga automaticamente i nuovi campi aggiunti nel nostro schema e li aggiunga a DynamoDB con i tipi corretti:

```

{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key": {
    "id" : $util.dynamodb.toDynamoDBJson($util.autoId())
  },
  "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}

```

Nell'esempio precedente, utilizziamo l'`$util.dynamodb.toDynamoDBJson(...)` helper per prendere automaticamente l'id generato e convertirlo nella rappresentazione DynamoDB di un attributo stringa. Quindi prendiamo tutti gli argomenti e li convertiamo nelle loro rappresentazioni DynamoDB e li inviamo nel campo `attributeValues` del modello.

Sono disponibili due versioni di ogni helper: una versione che restituisce un oggetto (ad esempio, `$util.dynamodb.toString(...)`) e una versione che restituisce l'oggetto come stringa JSON (ad esempio, `$util.dynamodb.toStringJson(...)`). Nell'esempio precedente abbiamo usato la versione che restituisce i dati come stringa JSON. Se vuoi modificare l'oggetto prima che venga usato nel modello, puoi scegliere di restituire invece un oggetto, come illustrato di seguito:

```

{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key": {
    "id" : $util.dynamodb.toDynamoDBJson($util.autoId())
  },

  #set( $myFoo = $util.dynamodb.toMapValues($ctx.args) )
  #set( $myFoo.version = $util.dynamodb.toNumber(1) )
  #set( $myFoo.timestamp = $util.dynamodb.toString($util.time.nowISO8601()) )

  "attributeValues" : $util.toJson($myFoo)
}

```

```
}
```

Nell'esempio precedente, gli argomenti convertiti vengono restituiti come mappa invece di una stringa JSON, e vengono quindi aggiunti i campi `version` e `timestamp` prima di eseguirne l'output nel campo `attributeValues` del modello utilizzando `$util.toJson(...)`.

La versione JSON di ogni helper equivale al wrapping della versione non JSON in `$util.toJson(...)`. Ad esempio, le istruzioni seguenti sono identiche:

```
$util.toStringJson("Hello, World!")  
$util.toJson($util.toString("Hello, World!"))
```

A DynamoDB

Elenco di utilità ToDynamoDB

`$util.dynamodb.toDynamoDB(Object)` : Map

Strumento generale di conversione degli oggetti per DynamoDB che converte gli oggetti di input nella rappresentazione DynamoDB appropriata. Rappresenta alcuni tipi in un determinato modo. Ad esempio, usa elenchi ("L") invece di set ("SS", "NS", "BS"). Questo restituisce un oggetto che descrive il valore dell'attributo DynamoDB.

Esempio di stringa

```
Input:      $util.dynamodb.toDynamoDB("foo")  
Output:     { "S" : "foo" }
```

Esempio di numero

```
Input:      $util.dynamodb.toDynamoDB(12345)  
Output:     { "N" : 12345 }
```

Esempio booleano

```
Input:      $util.dynamodb.toDynamoDB(true)  
Output:     { "BOOL" : true }
```

Esempio di elenco

```

Input:      $util.dynamodb.toDynamoDB([ "foo", 123, { "bar" : "baz" } ])
Output:     {
              "L" : [
                { "S" : "foo" },
                { "N" : 123 },
                {
                  "M" : {
                    "bar" : { "S" : "baz" }
                  }
                }
              ]
            }

```

Esempio di mappa

```

Input:      $util.dynamodb.toDynamoDB({ "foo": "bar", "baz" : 1234, "beep":
[ "boop" ] })
Output:     {
              "M" : {
                "foo" : { "S" : "bar" },
                "baz" : { "N" : 1234 },
                "beep" : {
                  "L" : [
                    { "S" : "boop" }
                  ]
                }
              }
            }

```

`$util.dynamodb.toDynamoDBJson(Object) : String`

Uguale a `$util.dynamodb.toDynamoDB(Object) : Map`, ma restituisce il valore dell'attributo DynamoDB come stringa codificata JSON.

Utilità ToString

Elenco di utilità ToString

`$util.dynamodb.toString(String) : String`

Converte una stringa di input nel formato stringa DynamoDB. Questo restituisce un oggetto che descrive il valore dell'attributo DynamoDB.

```
Input:      $util.dynamodb.toString("foo")
Output:     { "S" : "foo" }
```

`$util.dynamodb.toStringJson(String) : Map`

Uguale a `$util.dynamodb.toString(String) : String`, ma restituisce il valore dell'attributo DynamoDB come stringa codificata JSON.

`$util.dynamodb.toStringSet(List<String>) : Map`

Converte un elenco con stringhe nel formato del set di stringhe DynamoDB. Questo restituisce un oggetto che descrive il valore dell'attributo DynamoDB.

```
Input:      $util.dynamodb.toStringSet([ "foo", "bar", "baz" ])
Output:     { "SS" : [ "foo", "bar", "baz" ] }
```

`$util.dynamodb.toStringSetJson(List<String>) : String`

Uguale a `$util.dynamodb.toStringSet(List<String>) : Map`, ma restituisce il valore dell'attributo DynamoDB come stringa codificata JSON.

ToNumber utils

Elenco di utilità ToNumber

`$util.dynamodb.toNumber(Number) : Map`

Converte un numero nel formato numerico DynamoDB. Questo restituisce un oggetto che descrive il valore dell'attributo DynamoDB.

```
Input:      $util.dynamodb.toNumber(12345)
Output:     { "N" : 12345 }
```

`$util.dynamodb.toNumberJson(Number) : String`

Uguale a `$util.dynamodb.toNumber(Number) : Map`, ma restituisce il valore dell'attributo DynamoDB come stringa codificata JSON.

`$util.dynamodb.toNumberSet(List<Number>) : Map`

Converte un elenco di numeri nel formato del set di numeri DynamoDB. Questo restituisce un oggetto che descrive il valore dell'attributo DynamoDB.

```
Input:      $util.dynamodb.toNumberSet([ 1, 23, 4.56 ])
Output:     { "NS" : [ 1, 23, 4.56 ] }
```

`$util.dynamodb.toNumberSetJson(List<Number>) : String`

Uguale a `$util.dynamodb.toNumberSet(List<Number>)` : Map, ma restituisce il valore dell'attributo DynamoDB come stringa codificata JSON.

ToBinary Utils

Elenco di utilità ToBinary

`$util.dynamodb.toBinary(String) : Map`

Converte i dati binari codificati come stringa base64 in formato binario DynamoDB. Questo restituisce un oggetto che descrive il valore dell'attributo DynamoDB.

```
Input:      $util.dynamodb.toBinary("foo")
Output:     { "B" : "foo" }
```

`$util.dynamodb.toBinaryJson(String) : String`

Uguale a `$util.dynamodb.toBinary(String)` : Map, ma restituisce il valore dell'attributo DynamoDB come stringa codificata JSON.

`$util.dynamodb.toBinarySet(List<String>) : Map`

Converte un elenco di dati binari codificati come stringhe base64 in formato set binario DynamoDB. Questo restituisce un oggetto che descrive il valore dell'attributo DynamoDB.

```
Input:      $util.dynamodb.toBinarySet([ "foo", "bar", "baz" ])
Output:     { "BS" : [ "foo", "bar", "baz" ] }
```

`$util.dynamodb.toBinarySetJson(List<String>) : String`

Uguale a `$util.dynamodb.toBinarySet(List<String>)` : Map, ma restituisce il valore dell'attributo DynamoDB come stringa codificata JSON.

ToBoolean utils

Elenco degli strumenti ToBoolean

\$util.dynamodb.toBoolean(Boolean) : Map

Converte un booleano nel formato booleano DynamoDB appropriato. Questo restituisce un oggetto che descrive il valore dell'attributo DynamoDB.

```
Input:      $util.dynamodb.toBoolean(true)
Output:     { "BOOL" : true }
```

\$util.dynamodb.toBooleanJson(Boolean) : String

Uguale a `$util.dynamodb.toBoolean(Boolean) : Map`, ma restituisce il valore dell'attributo DynamoDB come stringa codificata JSON.

ToNull utils

Elenco di utilità ToNull

\$util.dynamodb.toNull() : Map

Restituisce un valore null nel formato null di DynamoDB. Questo restituisce un oggetto che descrive il valore dell'attributo DynamoDB.

```
Input:      $util.dynamodb.toNull()
Output:     { "NULL" : null }
```

\$util.dynamodb.toNullJson() : String

Uguale a `$util.dynamodb.toNull() : Map`, ma restituisce il valore dell'attributo DynamoDB come stringa codificata JSON.

Utilità ToList

Elenco di utilità ToList

\$util.dynamodb.toList(List) : Map

Converte un elenco di oggetti nel formato elenco DynamoDB. Ogni elemento dell'elenco viene inoltre convertito nel formato DynamoDB appropriato. Rappresenta alcuni oggetti nidificati in un determinato modo. Ad esempio, usa elenchi ("L") invece di set ("SS", "NS", "BS"). Questo restituisce un oggetto che descrive il valore dell'attributo DynamoDB.

```
Input:      $util.dynamodb.toList([ "foo", 123, { "bar" : "baz" } ])
Output:     {
              "L" : [
                  { "S" : "foo" },
                  { "N" : 123 },
                  {
                      "M" : {
                          "bar" : { "S" : "baz" }
                      }
                  }
              ]
            }
```

\$util.dynamodb.toListJson(List) : String

Uguale a `$util.dynamodb.toList(List) : Map`, ma restituisce il valore dell'attributo DynamoDB come stringa codificata JSON.

Utilità ToMap

Elenco di utilità ToMap

\$util.dynamodb.toMap(Map) : Map

Converte una mappa nel formato di mappa DynamoDB. Ogni valore nella mappa viene inoltre convertito nel formato DynamoDB appropriato. Rappresenta alcuni oggetti nidificati in un determinato modo. Ad esempio, usa elenchi ("L") invece di set ("SS", "NS", "BS"). Questo restituisce un oggetto che descrive il valore dell'attributo DynamoDB.

```
Input:      $util.dynamodb.toMap({ "foo": "bar", "baz" : 1234, "beep": [ "boop" ] })
```



```
Output:  {
    "M" : {
      "foo" : { "S" : "bar" },
      "baz" : { "N" : 1234 },
      "beep" : {
        "L" : [
          { "S" : "boop" }
        ]
      }
    }
  }
```

`$util.dynamodb.toJson(Map) : String`

Uguale a `$util.dynamodb.toMap(Map) : Map`, ma restituisce il valore dell'attributo DynamoDB come stringa codificata JSON.

`$util.dynamodb.toMapValues(Map) : Map`

Crea una copia della mappa in cui ogni valore è stato convertito nel formato DynamoDB appropriato. Rappresenta alcuni oggetti nidificati in un determinato modo. Ad esempio, usa elenchi ("L") invece di set ("SS", "NS", "BS").

```
Input:    $util.dynamodb.toMapValues({ "foo": "bar", "baz" : 1234, "beep":
[ "boop" ] })
Output:   {
    "foo" : { "S" : "bar" },
    "baz" : { "N" : 1234 },
    "beep" : {
      "L" : [
        { "S" : "boop" }
      ]
    }
  }
```

Note

Questo è leggermente diverso dal fatto che restituisce solo il contenuto del valore dell'attributo DynamoDB, ma non l'intero valore dell'attributo `$util.dynamodb.toMap(Map) : Map` stesso. Ad esempio, le istruzioni seguenti sono identiche:

```
$util.dynamodb.toMapValues($map)
$util.dynamodb.toMap($map).get("M")
```

\$util.dynamodb.toMapValuesJson(Map) : String

Uguale a `$util.dynamodb.toMapValues(Map) : Map`, ma restituisce il valore dell'attributo DynamoDB come stringa codificata JSON.

Utilità S3Object

Elenco delle utilità di S3Object

\$util.dynamodb.toS3Object(String key, String bucket, String region) : Map

Converte la chiave, il bucket e la regione nella rappresentazione dell'oggetto DynamoDB S3. Questo restituisce un oggetto che descrive il valore dell'attributo DynamoDB.

```
Input:      $util.dynamodb.toS3Object("foo", "bar", region = "baz")
Output:     { "S" : "{ \"s3\" : { \"key\" : \"foo\", \"bucket\" : \"bar\", \"region\" : \"baz\" } }" }
```

\$util.dynamodb.toS3ObjectJson(String key, String bucket, String region) : String

Uguale a `$util.dynamodb.toS3Object(String key, String bucket, String region) : Map`, ma restituisce il valore dell'attributo DynamoDB come stringa codificata JSON.

\$util.dynamodb.toS3Object(String key, String bucket, String region, String version) : Map

Converte la chiave, il bucket, la regione e la versione opzionale nella rappresentazione dell'oggetto DynamoDB S3. Questo restituisce un oggetto che descrive il valore dell'attributo DynamoDB.

```
Input:      $util.dynamodb.toS3Object("foo", "bar", "baz", "beep")
Output:     { "S" : "{ \"s3\" : { \"key\" : \"foo\", \"bucket\" : \"bar\", \"region\" : \"baz\", \"version\" = \"beep\" } }" }
```

\$util.dynamodb.toS3ObjectJson(String key, String bucket, String region, String version) : String

Uguale a `$util.dynamodb.toS3Object(String key, String bucket, String region, String version) : Map`, ma restituisce il valore dell'attributo DynamoDB come stringa codificata JSON.

\$util.dynamodb.fromS3ObjectJson(String) : Map

Accetta il valore stringa di un oggetto DynamoDB S3 e restituisce una mappa che contiene la chiave, il bucket, la regione e la versione opzionale.

```
Input:      $util.dynamodb.fromS3ObjectJson({ "S" : "{ \"s3\" : { \"key\" : \"foo\",
  \"bucket\" : \"bar\", \"region\" : \"baz\", \"version\" = \"beep\" } }" })
Output:     { "key" : "foo", "bucket" : "bar", "region" : "baz", "version" :
  "beep" }
```

Aiutanti Amazon RDS in \$util.rds

Note

Ora supportiamo principalmente il runtime APPSYNC_JS e la relativa documentazione.
[Prendi in considerazione l'utilizzo del runtime APPSYNC_JS e delle relative guide qui.](#)

`$util.rds` contiene metodi di supporto che formattano le operazioni di Amazon RDS eliminando i dati estranei negli output dei risultati.

elenco di utilità \$util.rds

\$util.rds.toJsonString(String serializedSQLResult): String

Restituisce un `String` trasformando il formato di risultato operativo grezzo di Amazon Relational Database Service (Amazon RDS) Data API con stringhe in una stringa più concisa. La stringa restituita è un elenco serializzato di record SQL del set di risultati. Ogni record è rappresentato da una raccolta di coppie chiave-valore. Le chiavi sono i nomi di colonna corrispondenti.

Se l'istruzione corrispondente nell'input era una query SQL che causa una mutazione (ad esempio INSERT, UPDATE, DELETE), viene restituito un elenco vuoto. Ad esempio, la query `select * from Books limit 2` fornisce il risultato non elaborato dell'operazione Amazon RDS Data:

```
{
  "sqlStatementResults": [
    {
      "numberOfRecordsUpdated": 0,
      "records": [
        [
          {
            "stringValue": "Mark Twain"
          },
          {
            "stringValue": "Adventures of Huckleberry Finn"
          },
          {
            "stringValue": "978-1948132817"
          }
        ],
        [
          {
            "stringValue": "Jack London"
          },
          {
            "stringValue": "The Call of the Wild"
          },
          {
            "stringValue": "978-1948132275"
          }
        ]
      ],
      "columnMetadata": [
        {
          "isSigned": false,
          "isCurrency": false,
          "label": "author",
          "precision": 200,
          "typeName": "VARCHAR",
          "scale": 0,
          "isAutoIncrement": false,
          "isCaseSensitive": false,
          "schemaName": "",
          "tableName": "Books",
          "type": 12,
          "nullable": 0,
          "arrayBaseColumnType": 0,

```

```

        "name": "author"
      },
      {
        "isSigned": false,
        "isCurrency": false,
        "label": "title",
        "precision": 200,
        "typeName": "VARCHAR",
        "scale": 0,
        "isAutoIncrement": false,
        "isCaseSensitive": false,
        "schemaName": "",
        "tableName": "Books",
        "type": 12,
        "nullable": 0,
        "arrayBaseColumnType": 0,
        "name": "title"
      },
      {
        "isSigned": false,
        "isCurrency": false,
        "label": "ISBN-13",
        "precision": 15,
        "typeName": "VARCHAR",
        "scale": 0,
        "isAutoIncrement": false,
        "isCaseSensitive": false,
        "schemaName": "",
        "tableName": "Books",
        "type": 12,
        "nullable": 0,
        "arrayBaseColumnType": 0,
        "name": "ISBN-13"
      }
    ]
  }
]
}

```

Il `util.rds.toJsonString` è:

```

[
  {

```

```
"author": "Mark Twain",
"title": "Adventures of Huckleberry Finn",
"ISBN-13": "978-1948132817"
},
{
  "author": "Jack London",
  "title": "The Call of the Wild",
  "ISBN-13": "978-1948132275"
},
]
```

\$util.rds.toJsonObject(String serializedSQLResult): Object

È lo stesso di `$util.rds.toJsonString`, ma il risultato è un `JSONObject`.

Aiutanti HTTP in \$util.http

Note

Ora supportiamo principalmente il runtime `APPSYNC_JS` e la relativa documentazione. [Prendi in considerazione l'utilizzo del runtime APPSYNC_JS e delle relative guide qui.](#)

L'`$util.http` fornisce metodi di supporto che è possibile utilizzare per gestire i parametri di richiesta HTTP e aggiungere intestazioni di risposta.

Elenco di utilità \$util.http

\$util.http.copyHeaders(Map) : Map

Copia le intestazioni dalla mappa, escluse le seguenti intestazioni HTTP con restrizioni:

- transfer-encoding
- connection
- host
- expect
- keep-alive
- upgrade
- proxy-authenticate

- proxy-authorization
- te
- content-length

È possibile utilizzare questa utilità per inoltrare le intestazioni delle richieste all'endpoint HTTP downstream.

```
{
  ...
  "params": {
    ...
    "headers": $util.http.copyHeaders($ctx.request.headers),
    ...
  },
  ...
}
```

`$ util.http. addResponseHeader(String, oggetto)`

Aggiunge una singola intestazione personalizzata con il nome (`String`) e il valore (`Object`) della risposta. Si applicano le limitazioni seguenti:

- Oltre all'elenco delle intestazioni con restrizioni per `copyHeaders(Map)`, i nomi delle intestazioni non possono corrispondere a nessuno dei seguenti:
 - Access-Control-Allow-Credentials
 - Access-Control-Allow-Origin
 - Access-Control-Expose-Headers
 - Access-Control-Max-Age
 - Access-Control-Allow-Methods
 - Access-Control-Allow-Headers
 - Vary
 - Content-Type
- I nomi delle intestazioni non possono iniziare con i prefissi limitati o. `x-amzn-` `x-amz-`
- La dimensione delle intestazioni di risposta personalizzate non può superare i 4 KB. Sono inclusi i nomi e i valori delle intestazioni.
- È necessario definire ogni intestazione di risposta una volta per operazione GraphQL. Tuttavia, se definisci più volte un'intestazione personalizzata con lo stesso nome, nella risposta viene

visualizzata la definizione più recente. Tutte le intestazioni vengono conteggiate ai fini del limite di dimensione dell'intestazione indipendentemente dalla denominazione.

- Le intestazioni con un nome vuoto o limitato (`String`) o un valore nullo verranno ignorate e (`Object`) genereranno un `ResponseHeaderError` errore che viene aggiunto all'output dell'operazione. `errors`

```
export function request(ctx) {
  util.http.addResponseHeader('itemsCount', 7)
  util.http.addResponseHeader('render', ctx.args.render)
  return {}
}
```

`$util.http.addResponseHeaders(Map)`

Aggiunge più intestazioni di risposta alla risposta dalla mappa di nomi e valori specificata. (`String`) (`Object`) Le stesse limitazioni elencate per il `addResponseHeader(String, Object)` metodo si applicano anche a questo metodo.

```
export function request(ctx) {
  const headers = {
    headerInt: 12,
    headerString: 'stringValue',
    headerObject: {
      field1: 7,
      field2: 'string'
    }
  }
  util.http.addResponseHeaders(headers)
  return {}
}
```

Helper XML in `$util.xml`

Note

Ora supportiamo principalmente il runtime `APPSYNC_JS` e la relativa documentazione. [Prendi in considerazione l'utilizzo del runtime APPSYNC_JS e delle relative guide qui.](#)

`$util.xml` contiene metodi di supporto che possono semplificare la traduzione delle risposte XML in JSON o in un dizionario.

Elenco di utilità `$util.xml`

`$util.xml.toMap(String) : Map`

Converte una stringa XML in un dizionario.

Input:

```
<?xml version="1.0" encoding="UTF-8"?>
<posts>
<post>
  <id>1</id>
  <title>Getting started with GraphQL</title>
</post>
</posts>
```

Output (JSON representation):

```
{
  "posts":{
    "post":{
      "id":1,
      "title":"Getting started with GraphQL"
    }
  }
}
```

Input:

```
<?xml version="1.0" encoding="UTF-8"?>
<posts>
<post>
  <id>1</id>
  <title>Getting started with GraphQL</title>
</post>
<post>
  <id>2</id>
  <title>Getting started with AWS AppSync</title>
</post>
```

```
</posts>
```

Output (JSON representation):

```
{
  "posts": {
    "post": [
      {
        "id": 1,
        "title": "Getting started with GraphQL"
      },
      {
        "id": 2,
        "title": "Getting started with AWS AppSync"
      }
    ]
  }
}
```

\$util.xml.toJsonString(String) : String

Converte una stringa XML in una stringa JSON. È simile a `toMap`, tranne per il fatto che l'output è una stringa. Questa funzione è utile se si desidera convertire direttamente e restituire la risposta XML da un oggetto HTTP in formato JSON.

\$util.xml.toJsonString(String, Boolean) : String

Converte una stringa XML in una stringa JSON con un parametro booleano opzionale per determinare se si desidera codificare il JSON come stringa.

Aiutanti di trasformazione in `$util.transform`

Note

Ora supportiamo principalmente il runtime `APPSYNC_JS` e la relativa documentazione. [Prendi in considerazione l'utilizzo del runtime APPSYNC_JS e delle relative guide qui.](#)

`$util.transform` contiene metodi di supporto che semplificano l'esecuzione di operazioni complesse su fonti di dati, come le operazioni di filtro di Amazon DynamoDB.

aiutanti per la trasformazione

Elenco di utilità degli aiutanti di trasformazione

`$util.transform.toDynamoDBFilterExpression(Map) : Map`

Converte una stringa di input in un'espressione di filtro da utilizzare con DynamoDB.

Input:

```
$util.transform.toDynamoDBFilterExpression({
  "title":{
    "contains":"Hello World"
  }
})
```

Output:

```
{
  "expression" : "contains(#title, :title_contains)"
  "expressionNames" : {
    "#title" : "title",
  },
  "expressionValues" : {
    ":title_contains" : { "S" : "Hello World" }
  },
}
```

`$util.transform.toElasticsearchQueryDSL(Map) : Map`

Converte l'input specificato nella sua espressione OpenSearch Query DSL equivalente, restituendola come stringa JSON.

Input:

```
$util.transform.toElasticsearchQueryDSL({
  "upvotes":{
    "ne":15,
    "range":[
      10,
      20
    ]
  },
})
```

```

    "title":{
      "eq":"hihihi",
      "wildcard":"h*i"
    }
  })

```

Output:

```

{
  "bool":{
    "must":[
      {
        "bool":{
          "must":[
            {
              "bool":{
                "must_not":{
                  "term":{
                    "upvotes":15
                  }
                }
              }
            },
            {
              "range":{
                "upvotes":{
                  "gte":10,
                  "lte":20
                }
              }
            }
          ]
        }
      },
      {
        "bool":{
          "must":[
            {
              "term":{
                "title":"hihihi"
              }
            },
            {
              "wildcard":{
                "title":"h*i"
              }
            }
          ]
        }
      }
    ]
  }
}

```

```
}
}
}
}
}
}
}
}
```

Si presume che l'operatore predefinito sia AND.

Transformation Helpers, filtri di sottoscrizione

Elenco delle utilità dei filtri di sottoscrizione di Transformation Helpers

```
$util.transform.toSubscriptionFilter(Map) : Map
```

Converte un oggetto Map di input in un SubscriptionFilter oggetto espressione. Il `$util.transform.toSubscriptionFilter` metodo viene utilizzato come input per l'`$extensions.setSubscriptionFilter()` estensione. Per ulteriori informazioni, consulta [Estensioni](#).

```
$util.transform.toSubscriptionFilter(Map, List) : Map
```

Converte un oggetto Map di input in un oggetto SubscriptionFilter espressione. Il `$util.transform.toSubscriptionFilter` metodo viene utilizzato come input per l'`$extensions.setSubscriptionFilter()` estensione. Per ulteriori informazioni, consulta [Estensioni](#).

Il primo argomento è l'oggetto Map di input che viene convertito nell'oggetto SubscriptionFilter espressione. Il secondo argomento riguarda i nomi List di campo che vengono ignorati nel primo oggetto Map di input durante la costruzione dell'oggetto SubscriptionFilter espressione.

```
$util.transform.toSubscriptionFilter(Map, List, Map) : Map
```

Converte un oggetto Map di input in un SubscriptionFilter oggetto espressione. Il `$util.transform.toSubscriptionFilter` metodo viene utilizzato come input per l'`$extensions.setSubscriptionFilter()` estensione. Per ulteriori informazioni, consulta [Estensioni](#).

Il primo argomento è l'oggetto di Map input che viene convertito nell'oggetto `SubscriptionFilter` espressione, il secondo argomento riguarda i nomi `List` di campo che verranno ignorati nel primo oggetto di Map input e il terzo argomento è un oggetto di Map input con regole rigorose che viene incluso durante la costruzione dell'oggetto `SubscriptionFilter` espressione. Queste regole rigorose sono incluse nell'oggetto `SubscriptionFilter` espressione in modo tale che almeno una delle regole venga soddisfatta per passare il filtro di sottoscrizione.

Argomenti del filtro di iscrizione

La tabella seguente spiega come vengono definiti gli argomenti delle seguenti utilità:

- `$util.transform.toSubscriptionFilter(Map) : Map`
- `$util.transform.toSubscriptionFilter(Map, List) : Map`
- `$util.transform.toSubscriptionFilter(Map, List, Map) : Map`

Argument 1: Map

L'argomento 1 è un Map oggetto con i seguenti valori chiave:

- nomi di campo
- «e»
- «o»

Per i nomi di campo utilizzati come chiavi, le condizioni relative alle voci di questi campi sono nel formato di. `"operator" : "value"`

L'esempio seguente mostra come aggiungere voci a: Map

```
"field_name" : {  
    "operator1" : value  
}  
  
## We can have multiple conditions for the same field_name:  
  
"field_name" : {  
    "operator1" : value  
    "operator2" : value
```

```

      .
      .
      .
    }

```

Quando un campo contiene due o più condizioni, si considera che tutte queste condizioni utilizzino l'operazione OR.

L'input Map può anche avere «e» e «or» come chiavi, il che implica che tutte le voci al suo interno devono essere unite utilizzando la logica AND o OR a seconda della chiave. I valori chiave «and» e «or» prevedono una serie di condizioni.

```

"and" : [
  {
    "field_name1" : {
      "operator1" : value
    }
  },
  {
    "field_name2" : {
      "operator1" : value
    }
  },
  :
  .
].

```

Nota che puoi annidare «and» e «or». Cioè, puoi aver annidato «e» /"or» all'interno di un altro blocco «e» /"or». Tuttavia, questo non funziona per campi semplici.

```

"and" : [
  {
    "field_name1" : {
      "operator" : value
    }
  },
  {
    "or" : [

```

```

        {
          "field_name2" : {
            "operator" : value
          }
        },

        {
          "field_name3" : {
            "operator" : value
          }
        }
      ]
    }.

```

L'esempio seguente mostra un input dell'argomento 1 utilizzando `$util.transform.toSubscriptionFilter(Map) : Map`.

Ingresso/i

Argomento 1: Mappa:

```

{
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "gt": 2000
      }
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    }
  ]
}

```



```
    },  
    {  
      "isPublished": {  
        "eq": false  
      }  
    }  
  ]  
}
```

Output

Il risultato è un Map oggetto:

```
{  
  "filterGroup": [  
    {  
      "filters": [  
        {  
          "fieldName": "percentageUp",  
          "operator": "lte",  
          "value": 50  
        },  
        {  
          "fieldName": "title",  
          "operator": "ne",  
          "value": "Book1"  
        },  
        {  
          "fieldName": "downvotes",  
          "operator": "gt",  
          "value": 2000  
        },  
        {  
          "fieldName": "author",  
          "operator": "eq",  
          "value": "Admin"  
        }  
      ]  
    },  
    {  
      "filters": [  
        {  
          "fieldName": "percentageUp",  
          "operator": "lte",
```

```
      "value": 50
    },
    {
      "fieldName": "title",
      "operator": "ne",
      "value": "Book1"
    },
    {
      "fieldName": "downvotes",
      "operator": "gt",
      "value": 2000
    },
    {
      "fieldName": "isPublished",
      "operator": "eq",
      "value": false
    }
  ]
},
{
  "filters": [
    {
      "fieldName": "percentageUp",
      "operator": "gte",
      "value": 20
    },
    {
      "fieldName": "title",
      "operator": "ne",
      "value": "Book1"
    },
    {
      "fieldName": "downvotes",
      "operator": "gt",
      "value": 2000
    },
    {
      "fieldName": "author",
      "operator": "eq",
      "value": "Admin"
    }
  ]
},
{
```

```

    "filters": [
      {
        "fieldName": "percentageUp",
        "operator": "gte",
        "value": 20
      },
      {
        "fieldName": "title",
        "operator": "ne",
        "value": "Book1"
      },
      {
        "fieldName": "downvotes",
        "operator": "gt",
        "value": 2000
      },
      {
        "fieldName": "isPublished",
        "operator": "eq",
        "value": false
      }
    ]
  }
]
}

```

Argument 2: List

L'argomento 2 contiene una `List` serie di nomi di campo che non devono essere considerati nell'input Map (argomento 1) durante la costruzione dell'oggetto `SubscriptionFilter` espressione. `List` Possono anche essere vuoti.

L'esempio seguente mostra gli input dell'argomento 1 e dell'argomento 2 utilizzando `$util.transform.toSubscriptionFilter(Map, List) : Map`.

Ingresso/i

Argomento 1: Mappa:

```

{
  "percentageUp": {
    "lte": 50,

```

```
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "gt": 20
      }
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

Argomento 2: Elenco:

```
["percentageUp", "author"]
```

Output

Il risultato è un Map oggetto:

```
{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "title",
          "operator": "ne",
```

```

        "value": "Book1"
      },
      {
        "fieldName": "downvotes",
        "operator": "gt",
        "value": 20
      },
      {
        "fieldName": "isPublished",
        "operator": "eq",
        "value": false
      }
    ]
  }
]
}

```

Argument 3: Map

L'argomento 3 è un Map oggetto che ha nomi di campo come valori chiave (non può avere «and» o «or»). Per i nomi di campo come chiavi, le condizioni in questi campi sono voci nel formato di "operator" : "value". A differenza dell'argomento 1, l'argomento 3 non può avere più condizioni nella stessa chiave. Inoltre, l'argomento 3 non contiene una clausola «and» o «or», quindi non è prevista nemmeno la nidificazione.

L'argomento 3 rappresenta un elenco di regole rigorose, che vengono aggiunte all'oggetto SubscriptionFilter espressione in modo che venga soddisfatta almeno una di queste condizioni per passare il filtro.

```

{
  "fieldname1": {
    "operator": value
  },
  "fieldname2": {
    "operator": value
  }
}
.
.
.

```

L'esempio seguente mostra gli input dell'argomento 1, dell'argomento 2 e dell'argomento 3 utilizzando `$util.transform.toSubscriptionFilter(Map, List, Map) : Map`.

Ingresso/i

Argomento 1: Mappa:

```
{
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "lt": 20
      }
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

Argomento 2: Elenco:

```
["percentageUp", "author"]
```

Argomento 3: Mappa:

```
{
  "upvotes": {
    "gte": 250
  },
  "author": {
    "eq": "Person1"
  }
}
```

Output

Il risultato è un Map oggetto:

```
{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 20
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
        },
        {
          "fieldName": "upvotes",
          "operator": "gte",
          "value": 250
        }
      ]
    },
    {
      "filters": [
        {
          "fieldName": "title",
```

```
      "operator": "ne",
      "value": "Book1"
    },
    {
      "fieldName": "downvotes",
      "operator": "gt",
      "value": 20
    },
    {
      "fieldName": "isPublished",
      "operator": "eq",
      "value": false
    },
    {
      "fieldName": "author",
      "operator": "eq",
      "value": "Person1"
    }
  ]
}
```

Aiutanti matematici in \$util.math

Note

Ora supportiamo principalmente il runtime APPSYNC_JS e la relativa documentazione. [Prendi in considerazione l'utilizzo del runtime APPSYNC_JS e delle relative guide qui.](#)

`$util.math` contiene metodi per facilitare le operazioni matematiche più comuni.

lista di utilità `$util.math`

`$util.math.roundNum(Double) : Integer`

Prende un valore doppio e lo arrotonda al numero intero più vicino.

`$util.math.minVal(Double, Double) : Double`

Richiede due doppi e restituisce il valore minimo tra i due doppi.

\$util.math.maxVal(Double, Double) : Double

Richiede due doppi e restituisce il valore massimo tra i due doppi.

\$util.math.randomDouble() : Double

Restituisce un doppio casuale compreso tra 0 e 1.

 Important

Questa funzione non deve essere utilizzata per nulla che richieda una casualità ad alta entropia (ad esempio, la crittografia).

\$util.math.randomWithinRange(Integer, Integer) : Integer

Restituisce un valore intero casuale all'interno dell'intervallo specificato, con il primo argomento che specifica il valore inferiore dell'intervallo e il secondo argomento che specifica il valore superiore dell'intervallo.

 Important

Questa funzione non deve essere utilizzata per nulla che richieda una casualità ad alta entropia (ad esempio, la crittografia).

Aiutanti per le stringhe in \$util.str

 Note

Ora supportiamo principalmente il runtime APPSYNC_JS e la relativa documentazione. [Prendi in considerazione l'utilizzo del runtime APPSYNC_JS e delle relative guide qui.](#)

\$util.str contiene metodi per facilitare le operazioni più comuni sulle stringhe.

Elenco delle utilità di \$util.str

\$util.str.toUpper(String) : String

Prende una stringa e la converte interamente in lettere maiuscole.

\$util.str.toLowerCase(String) : String

Prende una stringa e la converte interamente in lettere minuscole.

\$util.str.replace(String, String, String) : String

Sostituisce una sottostringa all'interno di una stringa con un'altra stringa. Il primo argomento specifica la stringa su cui eseguire l'operazione di sostituzione. Il secondo argomento specifica la sottostringa da sostituire. Il terzo argomento specifica la stringa con cui sostituire il secondo argomento. Di seguito è riportato un esempio di utilizzo di questa utilità:

```
INPUT:      $util.str.replace("hello world", "hello", "mellow")
OUTPUT:     "mellow world"
```

\$util.str.normalize(String, String) : String

Normalizza una stringa utilizzando uno dei quattro moduli di normalizzazione Unicode: NFC, NFD, NFKC o NFKD. Il primo argomento è la stringa da normalizzare. Il secondo argomento è «nfc», «nfd», «nfkc» o «nfkd» e specifica il tipo di normalizzazione da utilizzare per il processo di normalizzazione.

Estensioni

Note

Ora supportiamo principalmente il runtime APPSYNC_JS e la relativa documentazione.

[Prendi in considerazione l'utilizzo del runtime APPSYNC_JS e delle relative guide qui.](#)

\$extensionscontiene una serie di metodi per eseguire azioni aggiuntive all'interno dei resolver.

Estensioni di memorizzazione nella cache

\$extensions.evictFromApiCache(String, String, Object) : Object

Rimuove un elemento dalla cache lato server. AWS AppSync Il primo argomento è il nome del tipo. Il secondo argomento è il nome del campo. Il terzo argomento è un oggetto contenente elementi della coppia chiave-valore che specificano il valore della chiave di memorizzazione nella cache. È necessario inserire gli elementi nell'oggetto nello stesso ordine delle chiavi di memorizzazione nella cache del resolver memorizzato nella cache. cachingKey

Note

Questa utilità funziona solo per le mutazioni, non per le interrogazioni.

Estensioni di abbonamento

`$extensions.setSubscriptionFilter(filterJsonObject)`

Definisce filtri di abbonamento avanzati. Ogni evento di notifica di sottoscrizione viene valutato sulla base dei filtri di sottoscrizione forniti e invia notifiche ai clienti se tutti i filtri rispondono `true`. L'argomento è `filterJsonObject` descritto nella sezione seguente.

Note

È possibile utilizzare questo metodo di estensione solo nei modelli di mappatura delle risposte di un resolver di sottoscrizioni.

`$extensions.setSubscriptionInvalidationFilter(filterJsonObject)`

Definisce i filtri di invalidazione dell'abbonamento. I filtri di sottoscrizione vengono valutati in base al payload di invalidazione, quindi invalidano un determinato abbonamento se i filtri restituiscono lo stesso risultato. `true` L'argomento è descritto nella sezione `filterJsonObject` seguente.

Note

È possibile utilizzare questo metodo di estensione solo nei modelli di mappatura delle risposte di un resolver di sottoscrizioni.

`$extensions.invalidateSubscriptions(invalidationJsonObject)`

Utilizzato per avviare l'invalidazione dell'abbonamento a seguito di una mutazione. L'argomento è `invalidationJsonObject` descritto nella sezione seguente.

Note

Questa estensione può essere utilizzata solo nei modelli di mappatura delle risposte dei risolutori di mutazioni.

È possibile utilizzare al massimo cinque chiamate di `$extensions.invalidateSubscriptions()` metodo uniche in ogni singola richiesta. Se superi questo limite, riceverai un errore GraphQL.

Argomento: filterJsonObject

L'oggetto JSON definisce i filtri di sottoscrizione o di invalidazione. È una serie di filtri in un `filterGroup`. Ogni filtro è una raccolta di filtri individuali.

```
{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "userId",
          "operator": "eq",
          "value": 1
        }
      ]
    },
    {
      "filters": [
        {
          "fieldName": "group",
          "operator": "in",
          "value": ["Admin", "Developer"]
        }
      ]
    }
  ]
}
```

Ogni filtro ha tre attributi:

- `fieldName`— Il campo dello schema GraphQL.
- `operator`— Il tipo di operatore.
- `value`— I valori da confrontare con il `fieldName` valore di notifica dell'abbonamento.

Di seguito è riportato un esempio di assegnazione di questi attributi:

```
{
  "fieldName" : "severity",
  "operator" : "le",
  "value" : $context.result.severity
}
```

Campo: fieldName

Il tipo di stringa `fieldName` si riferisce a un campo definito nello schema GraphQL che corrisponde al `fieldName` payload di notifica della sottoscrizione. Quando viene trovata una corrispondenza, il `value` campo dello schema GraphQL viene confrontato con quello del filtro di notifica `value` della sottoscrizione. Nell'esempio seguente, il `fieldName` filtro corrisponde al `service` campo definito in un determinato tipo GraphQL. Se il payload di notifica contiene un `service` campo con un `value` equivalente a `AWS AppSync`, il filtro restituisce: `true`

```
{
  "fieldName" : "service",
  "operator" : "eq",
  "value" : "AWS AppSync"
}
```

Campo: valore

Il valore può essere di tipo diverso in base all'operatore:

- Un numero singolo o booleano
 - Esempi di stringhe: `"test"` `"service"`
 - Esempi di numeri: `12`, `45.75`
 - Esempi booleani: `true` `false`
- Coppie di numeri o stringhe
 - Esempio di coppia di stringhe: `["test1", "test2"]`, `["start", "end"]`
 - Esempio di coppia numerica: `[1, 4]`, `[67, 89]`, `[12.45, 95.45]`
- Matrici di numeri o stringhe
 - Esempio di array di stringhe: `["test1", "test2", "test3", "test4", "test5"]`
 - Esempio di array numerico: `[1, 2, 3, 4, 5]`, `[12.11, 46.13, 45.09, 12.54, 13.89]`

Campo: operatore

Una stringa con distinzione tra maiuscole e minuscole con i seguenti valori possibili:

Operatore	Description	Tipi di valori possibili
eq	Uguale	intero, float, stringa, booleano
uno	Non uguale	intero, float, stringa, booleano
leu	Minore di o uguale a	intero, float, stringa
lt	Less than	intero, float, stringa
età	Maggiore di o uguale a	intero, float, stringa
gt	Greater than	intero, float, stringa
contiene	Verifica la presenza di una sottosequenza o di un valore nel set.	numero intero, float, stringa
Non contiene	Verifica l'assenza di una sottosequenza o l'assenza di un valore nell'insieme.	numero intero, float, stringa
Inizia con	Verifica la presenza di un prefisso.	stringa
in	Verifica la presenza di elementi corrispondenti presenti nell'elenco.	Matrice di numeri interi, float o stringhe
notIn	Verifica la presenza di elementi corrispondenti che non sono presenti nell'elenco.	Matrice di numeri interi, float o stringhe
tra	Tra due valori	numero intero, float, stringa
Contiene Any	Contiene elementi comuni	numero intero, float, stringa

La tabella seguente descrive come ogni operatore viene utilizzato nella notifica di sottoscrizione.

eq (equal)

L'eqoperatore valuta `true` se il valore del campo di notifica della sottoscrizione corrisponde ed è strettamente uguale al valore del filtro. Nell'esempio seguente, il filtro valuta `true` se la notifica di sottoscrizione ha un `service` campo con il valore equivalente a. `AWS AppSync`

Tipi di valori possibili: integer, float, string, boolean

```
{
  "fieldName" : "service",
  "operator" : "eq",
  "value" : "AWS AppSync"
}
```

ne (not equal)

L'neoperatore valuta `true` se il valore del campo di notifica della sottoscrizione è diverso dal valore del filtro. Nell'esempio seguente, il filtro valuta `true` se la notifica di sottoscrizione ha un `service` campo con un valore diverso da. `AWS AppSync`

Tipi di valori possibili: integer, float, string, boolean

```
{
  "fieldName" : "service",
  "operator" : "ne",
  "value" : "AWS AppSync"
}
```

le (less or equal)

L'leoperatore valuta `true` se il valore del campo di notifica della sottoscrizione è inferiore o uguale al valore del filtro. Nell'esempio seguente, il filtro valuta `true` se la notifica di sottoscrizione contiene un `size` campo con un valore inferiore o uguale a. `5`

Tipi di valori possibili: integer, float, string

```
{
  "fieldName" : "size",
  "operator" : "le",
  "value" : 5
}
```

```
}
```

lt (less than)

L'`lt` operatore valuta `true` se il valore del campo di notifica della sottoscrizione è inferiore al valore del filtro. Nell'esempio seguente, il filtro valuta `true` se la notifica di sottoscrizione contiene un `size` campo con un valore inferiore a. 5

Tipi di valori possibili: integer, float, string

```
{
  "fieldName" : "size",
  "operator" : "lt",
  "value" : 5
}
```

ge (greater or equal)

L'`ge` operatore valuta `true` se il valore del campo di notifica della sottoscrizione è maggiore o uguale al valore del filtro. Nell'esempio seguente, il filtro valuta `true` se la notifica di sottoscrizione contiene un `size` campo con un valore maggiore o uguale a. 5

Tipi di valori possibili: integer, float, string

```
{
  "fieldName" : "size",
  "operator" : "ge",
  "value" : 5
}
```

gt (greater than)

L'`gt` operatore valuta `true` se il valore del campo di notifica della sottoscrizione è maggiore del valore del filtro. Nell'esempio seguente, il filtro valuta `true` se la notifica di sottoscrizione contiene un `size` campo con un valore maggiore di. 5

Tipi di valori possibili: integer, float, string

```
{
  "fieldName" : "size",
  "operator" : "gt",
}
```



```
"value" : 5
}
```

contains

L'`contains` operatore verifica la presenza di una sottostringa, di una sottosequenza o di un valore in un set o in un singolo elemento. Un filtro con l'`contains` operatore valuta `true` se il valore del campo di notifica della sottoscrizione contiene il valore del filtro. Nell'esempio seguente, il filtro valuta `true` se la notifica di sottoscrizione contiene un `seats` campo con il valore dell'array contenente il valore. `10`

Tipi di valori possibili: integer, float, string

```
{
  "fieldName" : "seats",
  "operator" : "contains",
  "value" : 10
}
```

In un altro esempio, il filtro valuta `true` se la notifica di sottoscrizione ha un `event` campo con `launch` come sottostringa.

```
{
  "fieldName" : "event",
  "operator" : "contains",
  "value" : "launch"
}
```

notContains

L'`notContains` operatore verifica l'assenza di una sottostringa, di una sottosequenza o di un valore in un set o in un singolo elemento. Il filtro con l'`notContains` operatore determina `true` se il valore del campo di notifica della sottoscrizione non contiene il valore del filtro. Nell'esempio seguente, il filtro valuta `true` se la notifica di sottoscrizione ha un `seats` campo il cui valore dell'array non contiene il valore. `10`

Tipi di valori possibili: integer, float, string

```
{
  "fieldName" : "seats",
```

```
"operator" : "notContains",  
"value" : 10  
}
```

In un altro esempio, il filtro valuta `true` se la notifica di sottoscrizione ha un valore di `event` campo senza `launch` come sottofondo.

```
{  
  "fieldName" : "event",  
  "operator" : "notContains",  
  "value" : "launch"  
}
```

`beginsWith`

L'`beginsWith` operatore verifica la presenza di un prefisso in una stringa. Il filtro contenente l'`beginsWith` operatore valuta `true` se il valore del campo di notifica della sottoscrizione inizia con il valore del filtro. Nell'esempio seguente, il filtro valuta `true` se la notifica di sottoscrizione contiene un `service` campo con un valore che inizia con `AWS`.

Tipo di valore possibile: stringa

```
{  
  "fieldName" : "service",  
  "operator" : "beginsWith",  
  "value" : "AWS"  
}
```

`in`

L'`in` operatore verifica la presenza di elementi corrispondenti in un array. Il filtro contenente l'`in` operatore valuta `true` se il valore del campo di notifica della sottoscrizione esiste in un array. Nell'esempio seguente, il filtro valuta `true` se la notifica di sottoscrizione ha un `severity` campo con uno dei valori presenti nell'array: `[1, 2, 3]`.

Tipo di valore possibile: Array of integer, float o string

```
{  
  "fieldName" : "severity",  
  "operator" : "in",  
  "value" : [1,2,3]  
}
```

```
}
```

notIn

L'`notIn` operatore verifica la presenza di elementi mancanti in un array. Il filtro contenente l'`notIn` operatore valuta `true` se il valore del campo di notifica della sottoscrizione non esiste nell'array. Nell'esempio seguente, il filtro valuta `true` se la notifica di sottoscrizione ha un `severity` campo con uno dei valori non presenti nell'array: `[1, 2, 3]`

Tipo di valore possibile: Array di numeri interi, float o string

```
{
  "fieldName" : "severity",
  "operator" : "notIn",
  "value" : [1,2,3]
}
```

between

L'`between` operatore verifica la presenza di valori tra due numeri o stringhe. Il filtro contenente l'`between` operatore valuta `true` se il valore del campo di notifica della sottoscrizione è compreso tra la coppia di valori del filtro. Nell'esempio seguente, il filtro valuta `true` se la notifica di sottoscrizione ha un `severity` campo con valori `2, 3, 4`

Tipi di valori possibili: coppia di numeri interi, float o string

```
{
  "fieldName" : "severity",
  "operator" : "between",
  "value" : [1,5]
}
```

containsAny

L'`containsAny` operatore verifica la presenza di elementi comuni negli array. Un filtro con l'`containsAny` operatore valuta `true` se l'intersezione tra il valore del set del campo di notifica della sottoscrizione e il valore del set di filtri non è vuota. Nell'esempio seguente, il filtro valuta `true` se la notifica di sottoscrizione contiene un `seats` campo con un valore di matrice contenente `0, 10, 15`. Ciò significa che il filtro valuterebbe `true` se la notifica di sottoscrizione avesse un valore di `seats` campo pari `[10, 11]` o `[15, 20, 30]`.

Tipi di valori possibili: integer, float o string

```
{
  "fieldName" : "seats",
  "operator" : "containsAny",
  "value" : [10, 15]
}
```

Logica AND

È possibile combinare più filtri utilizzando la logica AND definendo più voci all'interno dell'`filters` oggetto dell'`filterGroup` array. Nell'esempio seguente, i filtri valutano `true` se la notifica di sottoscrizione ha un `userId` campo con un valore equivalente a 1 AND un valore di `group` campo pari `Admin` o uguale a `Developer`.

```
{
  "filterGroup": [
    {
      "filters" : [
        {
          "fieldName" : "userId",
          "operator" : "eq",
          "value" : 1
        },
        {
          "fieldName" : "group",
          "operator" : "in",
          "value" : ["Admin", "Developer"]
        }
      ]
    }
  ]
}
```

Logica OR

È possibile combinare più filtri utilizzando la logica OR definendo più oggetti di filtro all'interno dell'`filterGroup` array. Nell'esempio seguente, i filtri valutano `true` se la notifica di sottoscrizione ha un `userId` campo con un valore equivalente a 1 OR un valore di `group` campo uguale a `Admin` o `Developer`.

```
{
  "filterGroup": [
    {
      "filters" : [
        {
          "fieldName" : "userId",
          "operator" : "eq",
          "value" : 1
        }
      ]
    },
    {
      "filters" : [
        {
          "fieldName" : "group",
          "operator" : "in",
          "value" : ["Admin", "Developer"]
        }
      ]
    }
  ]
}
```

Eccezioni

Tieni presente che esistono diverse restrizioni per l'utilizzo dei filtri:

- Nell'`filters` oggetto, possono esserci un massimo di cinque `fieldName` elementi unici per filtro. Ciò significa che è possibile combinare un massimo di cinque `fieldName` oggetti singoli utilizzando la logica AND.
- L'`containsAny` operatore può disporre di un massimo di venti valori.
- Possono esserci un massimo di cinque valori per gli `notIn` operatori `in` and.
- Ogni stringa può contenere un massimo di 256 caratteri.
- Ogni confronto tra stringhe distingue tra maiuscole e minuscole.
- Il filtraggio degli oggetti annidati consente fino a cinque livelli di filtraggio annidati.
- Ciascuno `filterGroup` può avere un massimo di 10 `filters`. Ciò significa che è possibile combinarne un massimo di 10 `filters` utilizzando la logica OR.

- L'inoperatore è un caso speciale della logica OR. Nell'esempio seguente, ce ne sono due filters:

```
{
  "filterGroup": [
    {
      "filters" : [
        {
          "fieldName" : "userId",
          "operator" : "eq",
          "value" : 1
        },
        {
          "fieldName" : "group",
          "operator" : "in",
          "value" : ["Admin", "Developer"]
        }
      ]
    }
  ]
}
```

Il gruppo di filtri precedente viene valutato come segue e conta ai fini del limite massimo di filtri:

```
{
  "filterGroup": [
    {
      "filters" : [
        {
          "fieldName" : "userId",
          "operator" : "eq",
          "value" : 1
        },
        {
          "fieldName" : "group",
          "operator" : "eq",
          "value" : "Admin"
        }
      ]
    },
    {
      "filters" : [
```

```

        {
          "fieldName" : "userId",
          "operator" : "eq",
          "value" : 1
        },
        {
          "fieldName" : "group",
          "operator" : "eq",
          "value" : "Developer"
        }
      ]
    }
  ]
}
```

Argomento: invalidationJsonObject

`invalidationJsonObject` Definisce quanto segue:

- **subscriptionField**— L'abbonamento allo schema GraphQL da invalidare. Un singolo abbonamento, definito come una stringa `subscriptionField`, viene considerato invalidato.
- **payload**— Un elenco di coppie chiave-valore che viene utilizzato come input per invalidare le sottoscrizioni se il filtro di invalidazione valuta in base ai relativi valori. `true`

L'esempio seguente invalida i client sottoscritti e connessi che utilizzano l'abbonamento quando il filtro di invalidazione definito nel resolver di `onUserDelete` sottoscrizione restituisce un risultato conforme al valore. `true` `payload`

```

$extensions.invalidateSubscriptions({
  "subscriptionField": "onUserDelete",
  "payload": {
    "group": "Developer"
    "type" : "Full-Time"
  }
})
```

AWS AppSync riferimento al modello di mappatura dei resolver per DynamoDB

Note

Ora supportiamo principalmente il runtime APPSYNC_JS e la relativa documentazione.
[Prendi in considerazione l'utilizzo del runtime APPSYNC_JS e delle relative guide qui.](#)

La funzione AWS AppSync DynamoDB ti consente di utilizzare [GraphQL](#) per archiviare e recuperare dati nelle tabelle Amazon DynamoDB esistenti nel tuo account mappando una richiesta GraphQL in entrata in una chiamata DynamoDB e quindi mappando la risposta DynamoDB a GraphQL. Questa sezione descrive i gestori di richieste e risposte per le operazioni DynamoDB supportate:

- [GetItem](#)- La GetItem richiesta consente di indicare alla funzione DynamoDB di effettuare una GetItem richiesta a DynamoDB e di specificare la chiave dell'elemento in DynamoDB e se utilizzare o meno una lettura coerente.
- [PutItem](#) - Il documento di mappatura della PutItem richiesta consente di indicare alla funzione DynamoDB di PutItem effettuare una richiesta a DynamoDB e di specificare la chiave dell'elemento in DynamoDB, il contenuto completo dell'elemento (composto da key e attributeValues) e le condizioni per il successo dell'operazione.
- [UpdateItem](#) - La UpdateItem richiesta consente di indicare alla funzione DynamoDB di effettuare una UpdateItem richiesta a DynamoDB e consente di specificare la chiave dell'elemento in DynamoDB, un'espressione di aggiornamento che descrive come aggiornare l'elemento in DynamoDB e le condizioni per il successo dell'operazione.
- [DeleteItem](#) - La DeleteItem richiesta consente di indicare alla funzione DynamoDB di effettuare una DeleteItem richiesta a DynamoDB e di specificare la chiave dell'elemento in DynamoDB e le condizioni per il successo dell'operazione.
- [Query](#): l'oggetto Query request consente di indicare al resolver DynamoDB di effettuare una richiesta Query a DynamoDB e di specificare l'espressione chiave, l'indice da utilizzare, i filtri aggiuntivi, il numero di elementi da restituire, se utilizzare letture coerenti, la direzione della query (avanti o indietro) e i token di impaginazione.
- [Scansione](#): la richiesta Scan consente di indicare alla funzione DynamoDB di effettuare una richiesta di scansione a DynamoDB e di specificare un filtro per escludere i risultati, quale indice

utilizzare, quanti elementi restituire, se utilizzare letture coerenti, token di impaginazione e scansioni parallele.

- [Sincronizzazione](#): l'oggetto di richiesta di sincronizzazione consente di recuperare tutti i risultati da una tabella DynamoDB e quindi ricevere solo i dati modificati dall'ultima query (gli aggiornamenti delta). Le richieste di sincronizzazione possono essere effettuate solo su sorgenti dati DynamoDB con versione. È possibile specificare un filtro per escludere i risultati, il numero di elementi da restituire, i token di paginazione e quando è stata avviata l'ultima operazione di sincronizzazione.
- [BatchGetItem](#) - L'oggetto BatchGetItem request consente di indicare alla funzione DynamoDB di effettuare una BatchGetItem richiesta a DynamoDB per recuperare più elementi, potenzialmente su più tabelle. Per questo oggetto di richiesta, è necessario specificare i nomi delle tabelle da cui recuperare gli elementi e le chiavi degli elementi da recuperare da ciascuna tabella.
- [BatchDeleteItem](#) - L'oggetto BatchDeleteItem request consente di indicare alla funzione DynamoDB di fare una BatchWriteItem richiesta a DynamoDB per eliminare più elementi, potenzialmente su più tabelle. Per questo oggetto di richiesta, è necessario specificare i nomi delle tabelle da cui eliminare gli elementi e le chiavi degli elementi da eliminare da ciascuna tabella.
- [BatchPutItem](#) - L'oggetto BatchPutItem request consente di indicare alla funzione DynamoDB di fare una BatchWriteItem richiesta a DynamoDB per inserire più elementi, potenzialmente su più tabelle. Per questo oggetto di richiesta, è necessario specificare i nomi delle tabelle in cui inserire gli elementi e gli elementi completi da inserire in ogni tabella.
- [TransactGetItems](#) - L'oggetto TransactGetItems request consente di indicare alla funzione DynamoDB di effettuare una TransactGetItems richiesta a DynamoDB per recuperare più elementi, potenzialmente su più tabelle. Per questo oggetto di richiesta, è necessario specificare il nome della tabella di ogni elemento della richiesta da cui recuperare l'elemento e la chiave di ogni elemento della richiesta da recuperare da ogni tabella.
- [TransactWriteItems](#) - L'oggetto TransactWriteItems request consente di indicare alla funzione DynamoDB di TransactWriteItems richiedere a DynamoDB di scrivere più elementi, potenzialmente su più tabelle. Per questo oggetto di richiesta, è necessario specificare il nome della tabella di destinazione di ogni elemento della richiesta, l'operazione di ogni elemento della richiesta da eseguire e la chiave di ogni elemento della richiesta da scrivere.
- [Sistema dei tipi \(mappatura delle richieste\)](#): scopri di più su come la digitazione DynamoDB è integrata nelle richieste. AWS AppSync
- [Sistema di tipi \(mappatura delle risposte\)](#): scopri di più su come i tipi DynamoDB vengono convertiti automaticamente in GraphQL o JSON in un payload di risposta.
- [Filtri](#): scopri di più sui filtri per le operazioni di interrogazione e scansione.

- [Espressioni](#) condizionali: scopri di più sulle espressioni di PutItem condizione e DeleteItem sulle operazioni. UpdateItem
- [Espressioni delle condizioni delle transazioni](#): ulteriori informazioni sulle espressioni di condizione per TransactWriteItems le operazioni.
- [Proiezioni](#): ulteriori informazioni su come specificare gli attributi nelle operazioni di lettura.

GetItem

Il documento di mappatura delle GetItem richieste consente di indicare al AWS AppSync resolver DynamoDB di effettuare una GetItem richiesta a DynamoDB e consente di specificare:

- La chiave dell'elemento in DynamoDB
- Se utilizzare una lettura consistente o no

Il documento di mappatura GetItem ha la seguente struttura:

```
{
  "version" : "2017-02-28",
  "operation" : "GetItem",
  "key" : {
    "foo" : ... typed value,
    "bar" : ... typed value
  },
  "consistentRead" : true,
  "projection" : {
    ...
  }
}
```

I campi sono definiti come segue:

GetItem campi

GetItem elenco dei campi

version

La versione di definizione del modello. Al momento sono supportate le versioni 2017-02-28 e 2018-05-29. Questo valore è obbligatorio.

operation

L'operazione DynamoDB da eseguire. Per eseguire l'operazione `GetItem` DynamoDB, il valore deve essere impostato su `GetItem`. Questo valore è obbligatorio.

key

La chiave dell'elemento in DynamoDB. Gli elementi DynamoDB possono avere una sola chiave hash o una chiave hash e una chiave di ordinamento, a seconda della struttura della tabella. Per ulteriori informazioni su come specificare un «valore digitato», consulta [Sistema di tipi \(mappatura delle richieste\)](#). Questo valore è obbligatorio.

consistentRead

Se eseguire o meno una lettura fortemente coerente con DynamoDB. Si tratta di un'opzione facoltativa, impostata di default su `false`.

projection

Una proiezione utilizzata per specificare gli attributi da restituire dall'operazione DynamoDB. [Per ulteriori informazioni sulle proiezioni, vedere Proiezioni](#). Questo campo è facoltativo.

L'elemento restituito da DynamoDB viene automaticamente convertito in tipi primitivi GraphQL e JSON ed è disponibile nel contesto di mappatura (`$.context.result`).

Per ulteriori informazioni sulla conversione dei tipi in DynamoDB, [vedere Sistema dei tipi](#) (mappatura delle risposte).

Per ulteriori informazioni sui modelli di mappatura delle risposte, consulta [Panoramica dei modelli di mappatura Resolver](#).

Esempio

L'esempio seguente è un modello di mappatura per una query `getThing(foo: String!, bar: String!)` GraphQL:

```
{
  "version" : "2017-02-28",
  "operation" : "GetItem",
  "key" : {
    "foo" : $util.dynamodb.toDynamoDBJson($ctx.args.foo),
```

```
    "bar" : $util.dynamodb.toDynamoDBJson($ctx.args.bar)
  },
  "consistentRead" : true
}
```

Per ulteriori informazioni sull'API `GetItem` di DynamoDB, consulta la [documentazione API di DynamoDB](#).

PutItem

Il documento di mappatura delle `PutItem` richieste consente di indicare al AWS AppSync resolver DynamoDB di effettuare una `PutItem` richiesta a DynamoDB e consente di specificare quanto segue:

- La chiave dell'elemento in DynamoDB
- L'intero contenuto della voce (costituita da `key` e `attributeValues`)
- Condizioni per la riuscita dell'operazione

Il documento di mappatura `PutItem` ha la seguente struttura:

```
{
  "version" : "2018-05-29",
  "operation" : "PutItem",
  "customPartitionKey" : "foo",
  "populateIndexFields" : boolean value,
  "key": {
    "foo" : ... typed value,
    "bar" : ... typed value
  },
  "attributeValues" : {
    "baz" : ... typed value
  },
  "condition" : {
    ...
  },
  "_version" : 1
}
```

I campi sono definiti come segue:

PutItem campi

PutItem elenco dei campi

version

La versione di definizione del modello. Al momento sono supportate le versioni 2017-02-28 e 2018-05-29. Questo valore è obbligatorio.

operation

L'operazione DynamoDB da eseguire. Per eseguire l'operazione PutItem DynamoDB, il valore deve essere impostato su PutItem. Questo valore è obbligatorio.

key

La chiave dell'elemento in DynamoDB. Gli elementi DynamoDB possono avere una sola chiave hash o una chiave hash e una chiave di ordinamento, a seconda della struttura della tabella. Per ulteriori informazioni su come specificare un «valore digitato», consulta [Sistema di tipi \(mappatura delle richieste\)](#). Questo valore è obbligatorio.

attributeValues

Gli altri attributi della voce da inserir in DynamoDB. Per ulteriori informazioni su come specificare un «valore digitato», vedete [Type system \(request mapping\)](#). Questo campo è facoltativo.

condition

Una condizione per determinare se la richiesta deve riuscire o no in base allo stato dell'oggetto già incluso in DynamoDB. Se non viene specificata alcuna condizione, la richiesta PutItem sovrascrive qualsiasi valore esistente per quella voce. Per ulteriori informazioni sulle condizioni, vedere Espressioni di [condizione](#). Questo valore è facoltativo.

_version

Valore numerico che rappresenta l'ultima versione nota di un elemento. Questo valore è facoltativo. Questo campo viene utilizzato per il rilevamento dei conflitti ed è supportato solo nelle origini dati con versione.

customPartitionKey

Se abilitato, questo valore di stringa modifica il formato dei ds_pk record ds_sk and utilizzati dalla tabella delta sync quando il controllo delle versioni è abilitato (per ulteriori informazioni, consulta [Conflict detection and sync](#) nella AWS AppSync Developer Guide). Se abilitata, è abilitata anche l'elaborazione della populateIndexFields voce. Questo campo è facoltativo.

populateIndexFields

Un valore booleano che, se abilitato insieme a `customPartitionKey`, crea nuove voci per ogni record nella tabella delta sync, in particolare nelle colonne `gsi_ds_pk` and `gsi_ds_sk`. Per ulteriori informazioni, consulta [Rilevamento e sincronizzazione dei conflitti](#) nella Guida per gli AWS AppSync sviluppatori. Questo campo è facoltativo.

L'elemento scritto in DynamoDB viene automaticamente convertito in tipi primitivi GraphQL e JSON ed è disponibile nel contesto di mappatura (`$context.result`

Per ulteriori informazioni sulla conversione dei tipi in DynamoDB, [vedere Sistema dei tipi](#) (mappatura delle risposte).

Per ulteriori informazioni sui modelli di mappatura delle risposte, consulta [Panoramica dei modelli di mappatura Resolver](#).

Esempio 1

L'esempio seguente è un modello di mappatura per una mutazione `updateThing(foo: String!, bar: String!, name: String!, version: Int!)` GraphQL.

Se non esiste alcuna voce con la chiave specificata, viene creata. Se esiste, viene sovrascritta.

```
{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key": {
    "foo" : $util.dynamodb.toDynamoDBJson($ctx.args.foo),
    "bar" : $util.dynamodb.toDynamoDBJson($ctx.args.bar)
  },
  "attributeValues" : {
    "name" : $util.dynamodb.toDynamoDBJson($ctx.args.name),
    "version" : $util.dynamodb.toDynamoDBJson($ctx.args.version)
  }
}
```

Esempio 2

L'esempio seguente è un modello di mappatura per una mutazione `updateThing(foo: String!, bar: String!, name: String!, expectedVersion: Int!)` GraphQL.

Questo esempio verifica che l'elemento attualmente in DynamoDB abbia `version` il campo impostato su `expectedVersion`

```
{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key": {
    "foo" : $util.dynamodb.toDynamoDBJson($ctx.args.foo),
    "bar" : $util.dynamodb.toDynamoDBJson($ctx.args.bar)
  },
  "attributeValues" : {
    "name"      : $util.dynamodb.toDynamoDBJson($ctx.args.name),
    #set( $newVersion = $context.arguments.expectedVersion + 1 )
    "version" : $util.dynamodb.toDynamoDBJson($newVersion)
  },
  "condition" : {
    "expression" : "version = :expectedVersion",
    "expressionValues" : {
      ":expectedVersion" : $util.dynamodb.toDynamoDBJson($expectedVersion)
    }
  }
}
```

Per ulteriori informazioni sull'API `PutItem` di DynamoDB, consulta la [documentazione API di DynamoDB](#).

UpdateItem

Il documento di mappatura delle `UpdateItem` richieste consente di indicare al AWS AppSync resolver DynamoDB di effettuare una `UpdateItem` richiesta a DynamoDB e consente di specificare quanto segue:

- La chiave dell'elemento in DynamoDB
- Un'espressione di aggiornamento che descrive come aggiornare l'elemento in DynamoDB
- Condizioni per la riuscita dell'operazione

Il documento di mappatura `UpdateItem` ha la seguente struttura:

```
{
  "version" : "2018-05-29",
```

```
"operation" : "UpdateItem",
"customPartitionKey" : "foo",
"populateIndexFields" : boolean value,
"key": {
  "foo" : ... typed value,
  "bar" : ... typed value
},
"update" : {
  "expression" : "someExpression",
  "expressionNames" : {
    "#foo" : "foo"
  },
  "expressionValues" : {
    ":bar" : ... typed value
  }
},
"condition" : {
  ...
},
"_version" : 1
}
```

I campi sono definiti come segue:

UpdateItem campi

UpdateItem elenco dei campi

version

La versione di definizione del modello. Al momento sono supportate le versioni 2017-02-28 e 2018-05-29. Questo valore è obbligatorio.

operation

L'operazione DynamoDB da eseguire. Per eseguire l'operazione UpdateItem DynamoDB, il valore deve essere impostato su UpdateItem. Questo valore è obbligatorio.

key

La chiave dell'elemento in DynamoDB. Gli elementi DynamoDB possono avere una sola chiave hash o una chiave hash e una chiave di ordinamento, a seconda della struttura della tabella.

[Per ulteriori informazioni sulla specificazione di un «valore digitato», consulta Sistema di tipi \(mappatura delle richieste\)](#). Questo valore è obbligatorio.

update

La `update` sezione consente di specificare un'espressione di aggiornamento che descrive come aggiornare l'elemento in DynamoDB. Per ulteriori informazioni su come scrivere espressioni di aggiornamento, consulta la documentazione di [UpdateExpressions DynamoDB](#). Questa sezione è obbligatoria.

La sezione `update` ha tre componenti:

expression

L'espressione di aggiornamento. Questo valore è obbligatorio.

expressionNames

Le sostituzioni per i segnaposto dell'attributo di espressione name sotto forma di coppie chiave-valore. La chiave corrisponde a un segnaposto per il nome utilizzato in `expression`, e il valore deve essere una stringa corrispondente al nome dell'attributo dell'elemento in DynamoDB. Questo è un campo facoltativo in cui vanno riportate solo le sostituzioni per i segnaposto dell'attributo di espressione name utilizzate in `expression`.

expressionValues

Le sostituzioni per i segnaposto dell'attributo di espressione value sotto forma di coppie chiave-valore. La chiave corrisponde a un segnaposto per un valore utilizzato in `expression`, mentre il valore deve essere un valore tipizzato. Per ulteriori informazioni su come specificare un «valore digitato», consulta [Type system](#) (request mapping). Questo elemento deve essere specificato. Questo è un campo facoltativo in cui vanno riportate solo le sostituzioni per i segnaposto dell'attributo di espressione value utilizzate in `expression`.

condition

Una condizione per determinare se la richiesta deve riuscire o no in base allo stato dell'oggetto già incluso in DynamoDB. Se non viene specificata alcuna condizione, la richiesta `UpdateItem` aggiorna qualsiasi valore esistente, indipendentemente dal suo stato attuale. Per ulteriori informazioni sulle condizioni, vedere Espressioni di [condizione](#). Questo valore è facoltativo.

_version

Valore numerico che rappresenta l'ultima versione nota di un elemento. Questo valore è facoltativo. Questo campo viene utilizzato per il rilevamento dei conflitti ed è supportato solo nelle origini dati con versione.

customPartitionKey

Se abilitato, questo valore di stringa modifica il formato dei ds_pk record ds_sk and utilizzati dalla tabella delta sync quando il controllo delle versioni è abilitato (per ulteriori informazioni, consulta [Conflict detection and sync](#) nella AWS AppSync Developer Guide). Se abilitata, è abilitata anche l'elaborazione della populateIndexFields voce. Questo campo è facoltativo.

populateIndexFields

Un valore booleano che, se abilitato insieme a customPartitionKey, crea nuove voci per ogni record nella tabella delta sync, in particolare nelle colonne gsi_ds_pk and gsi_ds_sk. Per ulteriori informazioni, consulta [Rilevamento e sincronizzazione dei conflitti](#) nella Guida per gli AWS AppSync sviluppatori. Questo campo è facoltativo.

L'elemento aggiornato in DynamoDB viene automaticamente convertito in tipi primitivi GraphQL e JSON ed è disponibile nel contesto di mappatura (`$context.result`

Per ulteriori informazioni sulla conversione dei tipi in DynamoDB, [vedere Sistema dei tipi](#) (mappatura delle risposte).

Per ulteriori informazioni sui modelli di mappatura delle risposte, consulta Panoramica dei modelli di mappatura [Resolver](#).

Esempio 1

L'esempio seguente è un modello di mappatura per la mutazione `upvote(id: ID!)` GraphQL.

In questo esempio, un elemento in DynamoDB ha i upvotes suoi campi `version` e incrementati di 1.

```
{
  "version" : "2017-02-28",
  "operation" : "UpdateItem",
  "key" : {
    "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
  },
  "update" : {
    "expression" : "ADD #voteField :plusOne, version :plusOne",
    "expressionNames" : {
      "#voteField" : "upvotes"
    },
  },
}
```

```

        "expressionValues" : {
            ":plusOne" : { "N" : 1 }
        }
    }
}

```

Esempio 2

L'esempio seguente è un modello di mappatura per una mutazione `updateItem(id: ID!, title: String, author: String, expectedVersion: Int!) GraphQL`.

Si tratta di un esempio complesso che verifica gli argomenti e genera dinamicamente l'espressione di aggiornamento in cui sono inclusi solo gli argomenti forniti dal client. Ad esempio, se `title` e `author` vengono omessi, non vengono aggiornati. Se viene specificato un argomento ma il suo valore è `null`, quel campo viene eliminato dall'oggetto in DynamoDB. Infine, l'operazione ha una condizione, che verifica se l'elemento attualmente in DynamoDB ha `version` il campo impostato su: `expectedVersion`

```

{
    "version" : "2017-02-28",

    "operation" : "UpdateItem",

    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    },

    ## Set up some space to keep track of things we're updating **
    #set( $expNames = {} )
    #set( $expValues = {} )
    #set( $expSet = {} )
    #set( $expAdd = {} )
    #set( $expRemove = [] )

    ## Increment "version" by 1 **
    ${!expAdd.put("version", ":newVersion")}
    ${!expValues.put(":newVersion", { "N" : 1 })}

    ## Iterate through each argument, skipping "id" and "expectedVersion" **
    #foreach( $entry in $context.arguments.entrySet() )
        #if( $entry.key != "id" && $entry.key != "expectedVersion" )
            #if( (!$entry.value) && ("${!entry.value}" == "") )

```

```

        ## If the argument is set to "null", then remove that attribute from
the item in DynamoDB **

        #set( $discard = ${expRemove.add("#${entry.key}")} )
        ${expNames.put("#${entry.key}", "${entry.key}")}
    #else
        ## Otherwise set (or update) the attribute on the item in DynamoDB **

        ${expSet.put("#${entry.key}", ":${entry.key}")}
        ${expNames.put("#${entry.key}", "${entry.key}")}

        #if( $entry.key == "ups" || $entry.key == "downs" )
            ${expValues.put(":${entry.key}", { "N" : $entry.value })}
        #else
            ${expValues.put(":${entry.key}", { "S" : "${entry.value}" })}
        #end
    #end
#end

## Start building the update expression, starting with attributes we're going to
SET **
#set( $expression = "" )
#if( !${expSet.isEmpty()} )
    #set( $expression = "SET" )
    #foreach( $entry in $expSet.entrySet() )
        #set( $expression = "${expression} ${entry.key} = ${entry.value}" )
        #if ( $foreach.hasNext )
            #set( $expression = "${expression}," )
        #end
    #end
#end

## Continue building the update expression, adding attributes we're going to ADD **
#if( !${expAdd.isEmpty()} )
    #set( $expression = "${expression} ADD" )
    #foreach( $entry in $expAdd.entrySet() )
        #set( $expression = "${expression} ${entry.key} ${entry.value}" )
        #if ( $foreach.hasNext )
            #set( $expression = "${expression}," )
        #end
    #end
#end
#end

```

```

    ## Continue building the update expression, adding attributes we're going to REMOVE
    **
    #if( !${expRemove.isEmpty()} )
        #set( $expression = "${expression} REMOVE" )

        #foreach( $entry in $expRemove )
            #set( $expression = "${expression} ${entry}" )
            #if ( $foreach.hasNext )
                #set( $expression = "${expression}," )
            #end
        #end
    #end

    ## Finally, write the update expression into the document, along with any
    expressionNames and expressionValues **
    "update" : {
        "expression" : "${expression}"
        #if( !${expNames.isEmpty()} )
            , "expressionNames" : $utils.toJson($expNames)
        #end
        #if( !${expValues.isEmpty()} )
            , "expressionValues" : $utils.toJson($expValues)
        #end
    },

    "condition" : {
        "expression" : "version = :expectedVersion",
        "expressionValues" : {
            ":expectedVersion" :
$util.dynamodb.toDynamoDBJson($ctx.args.expectedVersion)
        }
    }
}

```

Per ulteriori informazioni sull'API UpdateItem di DynamoDB, consulta la [documentazione API di DynamoDB](#).

DeleteItem

Il documento di mappatura delle DeleteItem richieste consente di indicare al AWS AppSync resolver DynamoDB di effettuare una DeleteItem richiesta a DynamoDB e consente di specificare quanto segue:

- La chiave dell'elemento in DynamoDB
- Condizioni per la riuscita dell'operazione

Il documento di mappatura `DeleteItem` ha la seguente struttura:

```
{
  "version" : "2018-05-29",
  "operation" : "DeleteItem",
  "customPartitionKey" : "foo",
  "populateIndexFields" : boolean value,
  "key": {
    "foo" : ... typed value,
    "bar" : ... typed value
  },
  "condition" : {
    ...
  },
  "_version" : 1
}
```

I campi sono definiti come segue:

DeleteItem campi

DeleteItem elenco dei campi

version

La versione di definizione del modello. Al momento sono supportate le versioni 2017-02-28 e 2018-05-29. Questo valore è obbligatorio.

operation

L'operazione DynamoDB da eseguire. Per eseguire l'operazione `DeleteItem` DynamoDB, il valore deve essere impostato su `DeleteItem`. Questo valore è obbligatorio.

key

La chiave dell'elemento in DynamoDB. Gli elementi DynamoDB possono avere una sola chiave hash o una chiave hash e una chiave di ordinamento, a seconda della struttura della tabella.

[Per ulteriori informazioni sulla specificazione di un «valore digitato», consulta Sistema di tipi \(mappatura delle richieste\)](#). Questo valore è obbligatorio.

condition

Una condizione per determinare se la richiesta deve riuscire o no in base allo stato dell'oggetto già incluso in DynamoDB. Se non viene specificata alcuna condizione, la richiesta `DeleteItem` elimina la voce indipendentemente dal suo stato attuale. [Per ulteriori informazioni sulle condizioni, vedere Espressioni di condizione](#). Questo valore è facoltativo.

_version

Valore numerico che rappresenta l'ultima versione nota di un elemento. Questo valore è facoltativo. Questo campo viene utilizzato per il rilevamento dei conflitti ed è supportato solo nelle origini dati con versione.

customPartitionKey

Se abilitato, questo valore di stringa modifica il formato dei `ds_pk` record `ds_sk` and utilizzati dalla tabella delta sync quando il controllo delle versioni è abilitato (per ulteriori informazioni, consulta [Conflict detection and sync](#) nella AWS AppSync Developer Guide). Se abilitata, è abilitata anche l'elaborazione della `populateIndexFields` voce. Questo campo è facoltativo.

populateIndexFields

Un valore booleano che, se abilitato insieme a `customPartitionKey`, crea nuove voci per ogni record nella tabella delta sync, in particolare nelle colonne `gsi_ds_pk` and `gsi_ds_sk`. Per ulteriori informazioni, consulta [Rilevamento e sincronizzazione dei conflitti](#) nella Guida per gli AWS AppSync sviluppatori. Questo campo è facoltativo.

L'elemento eliminato da DynamoDB viene automaticamente convertito in tipi primitivi GraphQL e JSON ed è disponibile nel contesto di mappatura (`$context.result`

Per ulteriori informazioni sulla conversione dei tipi in DynamoDB, [vedere Sistema dei tipi](#) (mappatura delle risposte).

Per ulteriori informazioni sui modelli di mappatura delle risposte, consulta [Panoramica dei modelli di mappatura Resolver](#).

Esempio 1

L'esempio seguente è un modello di mappatura per una mutazione `deleteItem(id: ID!)` GraphQL. Se esiste già una voce con questo ID, viene eliminata.

```
{
```

```
"version" : "2017-02-28",
"operation" : "DeleteItem",
"key" : {
  "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
}
}
```

Esempio 2

L'esempio seguente è un modello di mappatura per una mutazione `deleteItem(id: ID!, expectedVersion: Int!)` GraphQL. Se esiste già una voce con questo ID, viene eliminata, ma solo se il relativo campo `version` è impostato su `expectedVersion`:

```
{
  "version" : "2017-02-28",
  "operation" : "DeleteItem",
  "key" : {
    "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
  },
  "condition" : {
    "expression" : "attribute_not_exists(id) OR version = :expectedVersion",
    "expressionValues" : {
      ":expectedVersion" : $util.dynamodb.toDynamoDBJson($expectedVersion)
    }
  }
}
```

Per ulteriori informazioni sull'API `DeleteItem` di DynamoDB, consulta la [documentazione API di DynamoDB](#).

Query

Il documento di mappatura delle Query richiede di indicare al AWS AppSync resolver DynamoDB di effettuare una Query richiesta a DynamoDB e consente di specificare quanto segue:

- Espressione chiave
- Indice da utilizzare
- Eventuali filtri aggiuntivi
- Numero di voci da restituire
- Se utilizzare letture consistenti

- Direzione della query (avanti o indietro)
- Token di paginazione

Il documento di mappatura Query ha la seguente struttura:

```
{
  "version" : "2017-02-28",
  "operation" : "Query",
  "query" : {
    "expression" : "some expression",
    "expressionNames" : {
      "#foo" : "foo"
    },
    "expressionValues" : {
      ":bar" : ... typed value
    }
  },
  "index" : "fooIndex",
  "nextToken" : "a pagination token",
  "limit" : 10,
  "scanIndexForward" : true,
  "consistentRead" : false,
  "select" : "ALL_ATTRIBUTES" | "ALL_PROJECTED_ATTRIBUTES" | "SPECIFIC_ATTRIBUTES",
  "filter" : {
    ...
  },
  "projection" : {
    ...
  }
}
```

I campi sono definiti come segue:

Campi di interrogazione

Elenco dei campi di interrogazione

version

La versione di definizione del modello. Al momento sono supportate le versioni 2017-02-28 e 2018-05-29. Questo valore è obbligatorio.

operation

L'operazione DynamoDB da eseguire. Per eseguire l'operazione Query DynamoDB, il valore deve essere impostato su Query. Questo valore è obbligatorio.

query

La query sezione consente di specificare un'espressione di condizione chiave che descrive quali elementi recuperare da DynamoDB. Per ulteriori informazioni su come scrivere espressioni di condizioni chiave, consulta la documentazione di [KeyConditions DynamoDB](#). Questa sezione deve essere specificata.

expression

L'espressione della query. Questo campo deve essere specificato.

expressionNames

Le sostituzioni per i segnaposto dell'attributo di espressione name sotto forma di coppie chiave-valore. La chiave corrisponde a un segnaposto per il nome utilizzato in `expression`, e il valore deve essere una stringa corrispondente al nome dell'attributo dell'elemento in DynamoDB. Questo è un campo facoltativo in cui vanno riportate solo le sostituzioni per i segnaposto dell'attributo di espressione name utilizzate in `expression`.

expressionValues

Le sostituzioni per i segnaposto dell'attributo di espressione value sotto forma di coppie chiave-valore. La chiave corrisponde a un segnaposto per un valore utilizzato in `expression`, mentre il valore deve essere un valore tipizzato. Per ulteriori informazioni su come specificare un «valore digitato», consulta [Type system](#) (request mapping). Questo valore è obbligatorio. Questo è un campo facoltativo in cui vanno riportate solo le sostituzioni per i segnaposto dell'attributo di espressione value utilizzate in `expression`.

filter

Un ulteriore filtro che può essere utilizzato per filtrare i risultati da DynamoDB prima che siano restituiti. Per ulteriori informazioni sui filtri, consulta [Filtri](#). Questo campo è facoltativo.

index

Nome dell'indice su cui eseguire una query. L'operazione di interrogazione DynamoDB consente di eseguire la scansione degli indici secondari locali e degli indici secondari globali oltre all'indice della chiave primaria alla ricerca di una chiave hash. Se specificato, indica a DynamoDB di

interrogare l'indice specificato. Se omesso, la query viene eseguita sull'indice primario della chiave.

nextToken

Il token di paginazione utilizzato per continuare una query precedente, dalla quale deve essere ottenuto. Questo campo è facoltativo.

limit

Il numero massimo di item da valutare (non necessariamente il numero di item corrispondenti). Questo campo è facoltativo.

scanIndexForward

Un valore booleano che indica se la query deve essere eseguita in avanti o indietro. Si tratta di un campo facoltativo, impostato di default su `true`.

consistentRead

Un booleano che indica se utilizzare letture coerenti quando si esegue una query su DynamoDB. Si tratta di un campo facoltativo, impostato di default su `false`.

select

Per impostazione predefinita, il AWS AppSync resolver DynamoDB restituisce solo gli attributi proiettati nell'indice. Se sono necessari più attributi, puoi impostare questo campo. Questo campo è facoltativo. I valori supportati sono:

ALL_ATTRIBUTES

Restituisce tutti gli attributi della voce nella tabella o nell'indice specificati. Se si esegue una query su un indice secondario locale, DynamoDB recupera l'intero elemento dalla tabella principale per ogni elemento corrispondente nell'indice. Se l'indice è configurato per proiettare tutti gli attributi della voce, è possibile ottenere tutti i dati dall'indice secondario locale, senza necessità di recupero.

ALL_PROJECTED_ATTRIBUTES

Consentito solo durante l'esecuzione di una query su un indice. Recupera tutti gli attributi proiettati nell'indice. Se la configurazione dell'indice prevede che vi siano proiettati tutti gli attributi, il valore che restituisce è uguale a quello dato da `ALL_ATTRIBUTES`.

SPECIFIC_ATTRIBUTES

Restituisce solo gli attributi elencati negli. `projection expression` Questo valore restituito equivale a specificare i `projection expression` senza specificare alcun valore per.

Select

projection

Una proiezione utilizzata per specificare gli attributi da restituire dall'operazione DynamoDB. [Per ulteriori informazioni sulle proiezioni, vedere Proiezioni](#). Questo campo è facoltativo.

I risultati di DynamoDB vengono convertiti automaticamente in tipi primitivi GraphQL e JSON e sono disponibili nel contesto di mappatura (`()`). `$context.result`

Per ulteriori informazioni sulla conversione dei tipi in DynamoDB, [vedere Sistema dei tipi](#) (mappatura delle risposte).

Per ulteriori informazioni sui modelli di mappatura delle risposte, consulta [Panoramica dei modelli di mappatura Resolver](#).

I risultati hanno la struttura seguente:

```
{
  items = [ ... ],
  nextToken = "a pagination token",
  scannedCount = 10
}
```

I campi sono definiti come segue:

items

Un elenco contenente gli elementi restituiti dalla query DynamoDB.

nextToken

Se è possibile che ci siano altri risultati, `nextToken` contiene un token di paginazione utilizzabile in un'altra richiesta. Nota che AWS AppSync crittografa e offusca il token di impaginazione restituito da DynamoDB. In questo modo si evita che i dati della tabella siano inavvertitamente divulgati all'intermediario. Inoltre, i token di paginazione non possono essere utilizzati con più resolver diversi.

scannedCount

Il numero di voci corrispondenti all'espressione di condizione della query prima dell'applicazione di un'espressione di filtro (se presente).

Esempio

L'esempio seguente è un modello di mappatura per una query `getPosts(owner: ID!)` GraphQL.

In questo esempio, un indice secondario globale in una tabella viene interrogato per restituire tutti i post di proprietà dell'ID specificato.

```
{
  "version" : "2017-02-28",
  "operation" : "Query",
  "query" : {
    "expression" : "ownerId = :ownerId",
    "expressionValues" : {
      ":ownerId" : $util.dynamodb.toDynamoDBJson($context.arguments.owner)
    }
  },
  "index" : "owner-index"
}
```

Per ulteriori informazioni sull'API Query di DynamoDB, consulta la [documentazione API di DynamoDB](#).

Scan

Il documento di mappatura delle Scan richieste consente di indicare al AWS AppSync resolver DynamoDB di effettuare una Scan richiesta a DynamoDB e consente di specificare quanto segue:

- Un filtro per escludere i risultati
- Indice da utilizzare
- Numero di voci da restituire
- Se utilizzare letture consistenti
- Token di paginazione
- Scansioni parallele

Il documento di mappatura Scan ha la seguente struttura:

```
{
  "version" : "2017-02-28",
  "operation" : "Scan",
  "index" : "fooIndex",
  "limit" : 10,
  "consistentRead" : false,
  "nextToken" : "aPaginationToken",
  "totalSegments" : 10,
  "segment" : 1,
  "filter" : {
    ...
  },
  "projection" : {
    ...
  }
}
```

I campi sono definiti come segue:

Scansiona i campi

Elenco dei campi di scansione

version

La versione di definizione del modello. Al momento sono supportate le versioni 2017-02-28 e 2018-05-29. Questo valore è obbligatorio.

operation

L'operazione DynamoDB da eseguire. Per eseguire l'operazione Scan DynamoDB, il valore deve essere impostato su Scan. Questo valore è obbligatorio.

filter

Un filtro che può essere utilizzato per filtrare i risultati di DynamoDB prima che vengano restituiti. Per ulteriori informazioni sui filtri, consulta [Filtri](#). Questo campo è facoltativo.

index

Nome dell'indice su cui eseguire una query. L'operazione di interrogazione DynamoDB consente di eseguire la scansione degli indici secondari locali e degli indici secondari globali oltre all'indice

della chiave primaria alla ricerca di una chiave hash. Se specificato, indica a DynamoDB di interrogare l'indice specificato. Se omesso, la query viene eseguita sull'indice primario della chiave.

limit

Numero massimo di elementi da valutare in una sola volta. Questo campo è facoltativo.

consistentRead

Un valore booleano che indica se utilizzare letture coerenti quando si esegue una query su DynamoDB. Si tratta di un campo facoltativo, impostato di default su `false`.

nextToken

Il token di paginazione utilizzato per continuare una query precedente, dalla quale deve essere ottenuto. Questo campo è facoltativo.

select

Per impostazione predefinita, il AWS AppSync resolver DynamoDB restituisce solo gli attributi proiettati nell'indice. Se sono necessari più attributi, è possibile impostare questo campo. Questo campo è facoltativo. I valori supportati sono:

ALL_ATTRIBUTES

Restituisce tutti gli attributi della voce nella tabella o nell'indice specificati. Se si esegue una query su un indice secondario locale, DynamoDB recupera l'intero elemento dalla tabella principale per ogni elemento corrispondente nell'indice. Se l'indice è configurato per proiettare tutti gli attributi della voce, è possibile ottenere tutti i dati dall'indice secondario locale, senza necessità di recupero.

ALL_PROJECTED_ATTRIBUTES

Consentito solo durante l'esecuzione di una query su un indice. Recupera tutti gli attributi proiettati nell'indice. Se la configurazione dell'indice prevede che vi siano proiettati tutti gli attributi, il valore che restituisce è uguale a quello dato da `ALL_ATTRIBUTES`.

SPECIFIC_ATTRIBUTES

Restituisce solo gli attributi elencati negli. `projection expression` Questo valore restituito equivale a specificare i `projection expression` senza specificare alcun valore per. `Select`

totalSegments

Il numero di segmenti in cui partizionare la tabella durante una scansione parallela. Questo campo è facoltativo, ma deve essere specificato se è specificato anche `segment`.

segment

Il segmento della tabella in questa operazione quando si esegue una scansione parallela. Questo campo è facoltativo, ma deve essere specificato se è specificato anche `totalSegments`.

projection

Una proiezione utilizzata per specificare gli attributi da restituire dall'operazione DynamoDB. [Per ulteriori informazioni sulle proiezioni, vedere Proiezioni](#). Questo campo è facoltativo.

I risultati restituiti dalla scansione DynamoDB vengono convertiti automaticamente in tipi primitivi GraphQL e JSON e sono disponibili nel contesto di mappatura (`()`). `$context.result`

Per ulteriori informazioni sulla conversione dei tipi in DynamoDB, [vedere Sistema dei tipi](#) (mappatura delle risposte).

Per ulteriori informazioni sui modelli di mappatura delle risposte, consulta [Panoramica dei modelli di mappatura Resolver](#).

I risultati hanno la struttura seguente:

```
{
  items = [ ... ],
  nextToken = "a pagination token",
  scannedCount = 10
}
```

I campi sono definiti come segue:

items

Un elenco contenente gli elementi restituiti dalla scansione DynamoDB.

nextToken

Se potrebbero esserci più risultati, `nextToken` contiene un token di impaginazione che puoi utilizzare in un'altra richiesta. AWS AppSync crittografa e offusca il token di impaginazione

restituito da DynamoDB. In questo modo si evita che i dati della tabella siano inavvertitamente divulgati all'intermediario. Inoltre, questi token di paginazione non possono essere utilizzati con più resolver diversi.

scannedCount

Il numero di elementi recuperati da DynamoDB prima dell'applicazione di un'espressione di filtro (se presente).

Esempio 1

L'esempio seguente è un modello di mappatura per la query GraphQL: `allPosts`

In questo esempio, vengono restituite tutte le voci della tabella.

```
{
  "version" : "2017-02-28",
  "operation" : "Scan"
}
```

Esempio 2

L'esempio seguente è un modello di mappatura per la query GraphQL: `postsMatching(title: String!)`

In questo esempio, vengono restituite tutte le voci della tabella il cui titolo inizia con l'argomento `title`.

```
{
  "version" : "2017-02-28",
  "operation" : "Scan",
  "filter" : {
    "expression" : "begins_with(title, :title)",
    "expressionValues" : {
      ":title" : $util.dynamodb.toDynamoDBJson($context.arguments.title)
    },
  },
}
```

Per ulteriori informazioni sull'API Scan di DynamoDB, consulta la [documentazione API di DynamoDB](#).

Sync

Il documento di mappatura della Sync richiesta consente di recuperare tutti i risultati da una tabella DynamoDB e quindi ricevere solo i dati modificati dall'ultima query (gli aggiornamenti delta). Syncle richieste possono essere inoltrate solo a sorgenti dati DynamoDB con versione. È possibile specificare quanto segue:

- Un filtro per escludere i risultati
- Numero di voci da restituire
- Token di paginazione
- Quando è stata avviata l'ultima operazione Sync

Il documento di mappatura Sync ha la seguente struttura:

```
{
  "version" : "2018-05-29",
  "operation" : "Sync",
  "basePartitionKey": "Base Tables PartitionKey",
  "deltaIndexName": "delta-index-name",
  "limit" : 10,
  "nextToken" : "aPaginationToken",
  "lastSync" : 1550000000000,
  "filter" : {
    ...
  }
}
```

I campi sono definiti come segue:

Sincronizza campi

Elenco dei campi di sincronizzazione

version

La versione di definizione del modello. Al momento è supportato soltanto 2018-05-29. Questo valore è obbligatorio.

operation

L'operazione DynamoDB da eseguire. Per eseguire l'operazione Sync, il valore deve essere impostato su Sync. Questo valore è obbligatorio.

filter

Un filtro che può essere utilizzato per filtrare i risultati di DynamoDB prima che vengano restituiti. Per ulteriori informazioni sui filtri, consulta [Filtri](#). Questo campo è facoltativo.

limit

Numero massimo di elementi da valutare in una sola volta. Questo campo è facoltativo. Se omesso, il limite predefinito sarà impostato su 100 elementi. Il valore massimo per questo campo è 1000 articoli.

nextToken

Il token di paginazione utilizzato per continuare una query precedente, dalla quale deve essere ottenuto. Questo campo è facoltativo.

lastSync

Il momento, in millisecondi dall'epoca, in cui è iniziata l'ultima operazione Sync riuscita. Se specificato, vengono restituiti solo gli elementi che sono stati modificati dopo lastSync. Questo campo è facoltativo e deve essere compilato solo dopo aver recuperato tutte le pagine da un'operazione Sync iniziale. Se omesso, i risultati della tabella Base verranno restituiti, altrimenti verranno restituiti i risultati della tabella Delta.

basePartitionKey

La chiave di partizione della tabella Base utilizzata durante l'esecuzione di un'operazione. Sync Questo campo consente di eseguire un'Syncoperazione quando la tabella utilizza una chiave di partizione personalizzata. Questo campo è opzionale.

deltaIndexName

L'indice utilizzato per l'operazione. Sync Questo indice è necessario per abilitare un'Syncoperazione sull'intera tabella delta store quando la tabella utilizza una chiave di partizione personalizzata. L'Syncoperazione verrà eseguita sul GSI (creato su `gsi_ds_pk` and `gsi_ds_sk`). Questo campo è facoltativo.

I risultati restituiti dalla sincronizzazione DynamoDB vengono convertiti automaticamente in tipi primitivi GraphQL e JSON e sono disponibili nel contesto di mappatura (`$context.result`

Per ulteriori informazioni sulla conversione dei tipi in DynamoDB, [vedere Sistema dei tipi](#) (mappatura delle risposte).

Per ulteriori informazioni sui modelli di mappatura delle risposte, consulta [Panoramica dei modelli di mappatura Resolver](#).

I risultati hanno la struttura seguente:

```
{
  items = [ ... ],
  nextToken = "a pagination token",
  scannedCount = 10,
  startedAt = 1550000000000
}
```

I campi sono definiti come segue:

items

Un elenco contenente le voci restituite dalla sincronizzazione.

nextToken

Se potrebbero esserci più risultati, `nextToken` contiene un token di impaginazione che puoi utilizzare in un'altra richiesta. AWS AppSync crittografa e offusca il token di impaginazione restituito da DynamoDB. In questo modo si evita che i dati della tabella siano inavvertitamente divulgati all'intermediario. Inoltre, questi token di paginazione non possono essere utilizzati con più resolver diversi.

scannedCount

Il numero di elementi recuperati da DynamoDB prima dell'applicazione di un'espressione di filtro (se presente).

startedAt

Il momento, in millisecondi dall'epoca, in cui è iniziata l'operazione di sincronizzazione che è possibile memorizzare localmente e utilizzare in un'altra richiesta come argomento `lastSync`. Se un token di paginazione è stato incluso nella richiesta, questo valore sarà lo stesso di quello restituito dalla richiesta per la prima pagina di risultati.

Esempio

L'esempio seguente è un modello di mappatura per la query GraphQL: `syncPosts(nextToken: String, lastSync: AWSTimestamp)`

In questo esempio, se `lastSync` viene omissso, vengono restituite tutte le voci nella tabella di base. Se `lastSync` viene fornito, vengono restituite solo le voci nella tabella di sincronizzazione delta che sono state modificate dal momento in cui sono state sincronizzate `lastSync`.

```
{
  "version" : "2018-05-29",
  "operation" : "Sync",
  "limit": 100,
  "nextToken": $util.toJson($util.defaultIfNull($ctx.args.nextToken, null)),
  "lastSync": $util.toJson($util.defaultIfNull($ctx.args.lastSync, null))
}
```

BatchGetItem

Il documento di mappatura delle `BatchGetItem` richieste consente di indicare al AWS AppSync resolver DynamoDB di effettuare una `BatchGetItem` richiesta a DynamoDB per recuperare più elementi, potenzialmente su più tabelle. Per questo modello di richiesta, è necessario specificare quanto segue:

- I nomi delle tabelle da cui recuperare le voci
- Le chiavi delle voci da recuperare da ciascuna tabella

Si applicano i limiti `BatchGetItem` di DynamoDB e non si può inserire alcuna espressione di condizione.

Il documento di mappatura `BatchGetItem` ha la seguente struttura:

```
{
  "version" : "2018-05-29",
  "operation" : "BatchGetItem",
  "tables" : {
    "table1": {
      "keys": [
        ## Item to retrieve Key
      ]
    }
  }
}
```

```

        {
            "foo" : ... typed value,
            "bar" : ... typed value
        },
        ## Item2 to retrieve Key
        {
            "foo" : ... typed value,
            "bar" : ... typed value
        }
    ],
    "consistentRead": true|false,
    "projection" : {
        ...
    }
},
"table2": {
    "keys": [
        ## Item3 to retrieve Key
        {
            "foo" : ... typed value,
            "bar" : ... typed value
        },
        ## Item4 to retrieve Key
        {
            "foo" : ... typed value,
            "bar" : ... typed value
        }
    ],
    "consistentRead": true|false,
    "projection" : {
        ...
    }
}
}
}
}

```

I campi sono definiti come segue:

BatchGetItem campi

BatchGetItem elenco dei campi

version

La versione di definizione del modello. Solo 2018-05-29 è supportata. Questo valore è obbligatorio.

operation

L'operazione DynamoDB da eseguire. Per eseguire l'operazione BatchGetItem DynamoDB, il valore deve essere impostato su BatchGetItem. Questo valore è obbligatorio.

tables

Le tabelle DynamoDB da cui recuperare gli elementi. Il valore è una mappa in cui i nomi delle tabelle sono specificati come chiavi della mappa. Occorre specificare almeno una tabella. Questo valore `tables` è obbligatorio.

keys

Elenco di chiavi DynamoDB che rappresentano la chiave primaria degli elementi da recuperare. Gli elementi DynamoDB possono avere una sola chiave hash o una chiave hash e una chiave di ordinamento, a seconda della struttura della tabella. Per ulteriori informazioni su come specificare un «valore digitato», consulta [Sistema di tipi \(mappatura delle richieste\)](#).

consistentRead

Se utilizzare una lettura coerente durante l'esecuzione di un'GetItem operazione. Questo valore è opzionale e l'impostazione predefinita è `false`.

projection

Una proiezione utilizzata per specificare gli attributi da restituire dall'operazione DynamoDB. [Per ulteriori informazioni sulle proiezioni, vedere Proiezioni](#). Questo campo è facoltativo.

Aspetti da ricordare:

- Se una voce non è stata recuperata dalla tabella, un elemento null compare nel blocco di dati relativo a quella tabella.
- I risultati delle chiamate vengono ordinati per tabella, in base all'ordine in cui sono stati forniti all'interno del modello di mappatura della richiesta.

- Ogni Get comando all'interno di a BatchGetItem è atomico, tuttavia un batch può essere parzialmente elaborato. Se un batch viene elaborato parzialmente a causa di un errore, le chiavi non elaborate vengono restituite nell'ambito del risultato dell'invocazione all'interno del blocco unprocessedKeys.
- BatchGetItem ha un limite di 100 chiavi.

Per il seguente esempio di modello di mappatura della richiesta:

```
{
  "version": "2018-05-29",
  "operation": "BatchGetItem",
  "tables": {
    "authors": [
      {
        "author_id": {
          "S": "a1"
        }
      },
    ],
    "posts": [
      {
        "author_id": {
          "S": "a1"
        },
        "post_id": {
          "S": "p2"
        }
      },
    ],
  },
}
```

Il risultato dell'invocazione disponibile in `$ctx.result` è il seguente:

```
{
  "data": {
    "authors": [null],
    "posts": [
      # Was retrieved
      {
        "author_id": "a1",
```



```

        "post_id": "p2",
        "post_title": "title",
        "post_description": "description",
      }
    ]
  },
  "unprocessedKeys": {
    "authors": [
      # This item was not processed due to an error
      {
        "author_id": "a1"
      }
    ],
    "posts": []
  }
}

```

Il messaggio `$ctx.error` contiene dettagli relativi all'errore. Le chiavi dati, `unprocessedKeys` e ogni chiave di tabella disponibile nel modello di mappatura della richiesta sono sicuramente presenti nel risultato dell'invocazione. Le voci eliminate compaiono nel blocco dati. Le voci non elaborate vengono contrassegnate come null all'interno del blocco dati e vengono inserite nel blocco `unprocessedKeys`.

Per un esempio più completo, segui il tutorial su DynamoDB Batch [con AppSync questo Tutorial: DynamoDB batch resolvers](#).

BatchDeleteItem

Il documento di mappatura delle `BatchDeleteItem` richieste consente di indicare al AWS AppSync resolver DynamoDB di effettuare una `BatchWriteItem` richiesta a DynamoDB per eliminare più elementi, potenzialmente su più tabelle. Per questo modello di richiesta, è necessario specificare quanto segue:

- I nomi delle tabelle da cui eliminare le voci
- Le chiavi delle voci da eliminare da ciascuna tabella

Si applicano i limiti `BatchWriteItem` di DynamoDB e non si può inserire alcuna espressione di condizione.

Il documento di mappatura `BatchDeleteItem` ha la seguente struttura:

```
{
```

```
"version" : "2018-05-29",
"operation" : "BatchDeleteItem",
"tables" : {
  "table1": [
    ## Item to delete Key
    {
      "foo" : ... typed value,
      "bar" : ... typed value
    },
    ## Item2 to delete Key
    {
      "foo" : ... typed value,
      "bar" : ... typed value
    }
  ],
  "table2": [
    ## Item3 to delete Key
    {
      "foo" : ... typed value,
      "bar" : ... typed value
    },
    ## Item4 to delete Key
    {
      "foo" : ... typed value,
      "bar" : ... typed value
    }
  ],
}
```

I campi sono definiti come segue:

BatchDeleteItem campi

BatchDeleteItem elenco dei campi

version

La versione di definizione del modello. Solo 2018-05-29 è supportata. Questo valore è obbligatorio.

operation

L'operazione DynamoDB da eseguire. Per eseguire l'operazione BatchDeleteItem DynamoDB, il valore deve essere impostato su BatchDeleteItem. Questo valore è obbligatorio.

tables

Le tabelle DynamoDB da cui eliminare gli elementi. Ogni tabella è un elenco di chiavi DynamoDB che rappresentano la chiave primaria degli elementi da eliminare. Gli elementi DynamoDB possono avere una sola chiave hash o una chiave hash e una chiave di ordinamento, a seconda della struttura della tabella. Per ulteriori informazioni su come specificare un «valore digitato», consulta [Sistema di tipi \(mappatura delle richieste\)](#). Occorre specificare almeno una tabella. Il `tables` valore è obbligatorio.

Aspetti da ricordare:

- A differenza dell'operazione `DeleteItem`, nella risposta non viene restituita la voce completamente eliminata. Viene restituita solo la chiave passata.
- Se una voce non è stata eliminata dalla tabella, un elemento null compare nel blocco di dati relativo a quella tabella.
- I risultati delle chiamate vengono ordinati per tabella, in base all'ordine in cui sono stati forniti all'interno del modello di mappatura della richiesta.
- Ogni `Delete` comando all'interno di `a` è atomico. `BatchDeleteItem` Tuttavia, un batch può essere parzialmente elaborato. Se un batch viene elaborato parzialmente a causa di un errore, le chiavi non elaborate vengono restituite nell'ambito del risultato dell'invocazione all'interno del blocco `unprocessedKeys`.
- `BatchDeleteItem` ha un limite di 25 chiavi.
- Questa operazione non è supportata se utilizzata con il rilevamento dei conflitti. L'utilizzo di entrambi allo stesso tempo può causare un errore.

Per il seguente esempio di modello di mappatura della richiesta:

```
{
  "version": "2018-05-29",
  "operation": "BatchDeleteItem",
  "tables": {
    "authors": [
      {
        "author_id": {
          "S": "a1"
        }
      }
    ],
  },
}
```

```

    ],
    "posts": [
      {
        "author_id": {
          "S": "a1"
        },
        "post_id": {
          "S": "p2"
        }
      }
    ]
  },
}

```

Il risultato dell'invocazione disponibile in `$ctx.result` è il seguente:

```

{
  "data": {
    "authors": [null],
    "posts": [
      # Was deleted
      {
        "author_id": "a1",
        "post_id": "p2"
      }
    ]
  },
  "unprocessedKeys": {
    "authors": [
      # This key was not processed due to an error
      {
        "author_id": "a1"
      }
    ],
    "posts": []
  }
}

```

Il messaggio `$ctx.error` contiene dettagli relativi all'errore. Le chiavi dati, `unprocessedKeys` e ogni chiave di tabella disponibile nel modello di mappatura della richiesta sono sicuramente presenti nel risultato dell'invocazione. Le voci eliminate sono presenti nel blocco di dati. Le voci non

elaborate vengono contrassegnate come null all'interno del blocco dati e vengono inserite nel blocco `unprocessedKeys`.

Per un esempio più completo, segui il tutorial su DynamoDB Batch [con AppSync questo Tutorial: `DynamoDB` batch resolvers](#).

BatchPutItem

Il documento di mappatura delle `BatchPutItem` richieste consente di indicare al AWS AppSync resolver DynamoDB di fare una `BatchWriteItem` richiesta a DynamoDB per inserire più elementi, potenzialmente su più tabelle. Per questo modello di richiesta, è necessario specificare quanto segue:

- I nomi delle tabelle in cui inserire le voci
- Le voci complete da inserire in ciascuna tabella

Si applicano i limiti `BatchWriteItem` di DynamoDB e non si può inserire alcuna espressione di condizione.

Il documento di mappatura `BatchPutItem` ha la seguente struttura:

```
{
  "version" : "2018-05-29",
  "operation" : "BatchPutItem",
  "tables" : {
    "table1": [
      ## Item to put
      {
        "foo" : ... typed value,
        "bar" : ... typed value
      },
      ## Item2 to put
      {
        "foo" : ... typed value,
        "bar" : ... typed value
      }
    ],
    "table2": [
      ## Item3 to put
      {
        "foo" : ... typed value,
        "bar" : ... typed value
      }
    ]
  }
}
```

```
    },  
    ## Item4 to put  
    {  
        "foo" : ... typed value,  
        "bar" : ... typed value  
    }],  
  }  
}
```

I campi sono definiti come segue:

BatchPutItem campi

BatchPutItem elenco dei campi

version

La versione di definizione del modello. Solo 2018-05-29 è supportata. Questo valore è obbligatorio.

operation

L'operazione DynamoDB da eseguire. Per eseguire l'operazione BatchPutItem DynamoDB, il valore deve essere impostato su BatchPutItem. Questo valore è obbligatorio.

tables

Le tabelle DynamoDB in cui inserire gli elementi. Ogni voce della tabella rappresenta un elenco di elementi DynamoDB da inserire per questa tabella specifica. Occorre specificare almeno una tabella. Questo valore è obbligatorio.

Aspetti da ricordare:

- Se l'operazione va a buon fine, nella risposta vengono restituite le voci inserite completamente.
- Se una voce non è stata inserita nella tabella, un elemento null viene visualizzato nel blocco di dati relativo a quella tabella.
- Gli elementi inseriti vengono ordinati per tabella, in base all'ordine in cui sono stati forniti all'interno del modello di mappatura della richiesta.
- Ogni Put comando all'interno di a BatchPutItem è atomico, tuttavia un batch può essere parzialmente elaborato. Se un batch viene elaborato parzialmente a causa di un errore, le chiavi

non elaborate vengono restituite nell'ambito del risultato dell'invocazione all'interno del blocco `unprocessedKeys`.

- `BatchPutItem` ha un limite di 25 voci.
- Questa operazione non è supportata se utilizzata con il rilevamento dei conflitti. L'utilizzo di entrambi allo stesso tempo può causare un errore.

Per il seguente esempio di modello di mappatura della richiesta:

```
{
  "version": "2018-05-29",
  "operation": "BatchPutItem",
  "tables": {
    "authors": [
      {
        "author_id": {
          "S": "a1"
        },
        "author_name": {
          "S": "a1_name"
        }
      },
    ],
    "posts": [
      {
        "author_id": {
          "S": "a1"
        },
        "post_id": {
          "S": "p2"
        },
        "post_title": {
          "S": "title"
        }
      },
    ],
  },
}
```

Il risultato dell'invocazione disponibile in `$ctx.result` è il seguente:

```
{
```

```
"data": {
  "authors": [
    null
  ],
  "posts": [
    # Was inserted
    {
      "author_id": "a1",
      "post_id": "p2",
      "post_title": "title"
    }
  ]
},
"unprocessedItems": {
  "authors": [
    # This item was not processed due to an error
    {
      "author_id": "a1",
      "author_name": "a1_name"
    }
  ],
  "posts": []
}
```

Il messaggio `$ctx.error` contiene dettagli relativi all'errore. Le chiavi dati, `unprocessedItems` e ogni chiave di tabella disponibile nel modello di mappatura della richiesta sono sicuramente presenti nel risultato dell'invocazione. Le voci inserite si trovano nel blocco di dati. Le voci non elaborate vengono contrassegnate come null all'interno del blocco dati e vengono inserite nel blocco `unprocessedItems`.

Per un esempio più completo, segui il tutorial su DynamoDB Batch [con AppSync questo Tutorial: DynamoDB batch resolvers](#).

TransactGetItems

Il documento di mappatura delle `TransactGetItems` richieste consente di indicare al AWS AppSync resolver DynamoDB di effettuare una `TransactGetItems` richiesta a DynamoDB per recuperare più elementi, potenzialmente su più tabelle. Per questo modello di richiesta, è necessario specificare quanto segue:

- Il nome della tabella di ogni elemento di richiesta da cui recuperare l'elemento
- La chiave di ogni elemento di richiesta da recuperare da ogni tabella

Si applicano i limiti `TransactGetItems` di DynamoDB e non si può inserire alcuna espressione di condizione.

Il documento di mappatura `TransactGetItems` ha la seguente struttura:

```
{
  "version": "2018-05-29",
  "operation": "TransactGetItems",
  "transactItems": [
    ## First request item
    {
      "table": "table1",
      "key": {
        "foo": ... typed value,
        "bar": ... typed value
      },
      "projection" : {
        ...
      }
    },
    ## Second request item
    {
      "table": "table2",
      "key": {
        "foo": ... typed value,
        "bar": ... typed value
      },
      "projection" : {
        ...
      }
    }
  ]
}
```

I campi sono definiti come segue:

TransactGetItems campi

TransactGetItems elenco dei campi

version

La versione di definizione del modello. Solo 2018-05-29 è supportata. Questo valore è obbligatorio.

operation

L'operazione DynamoDB da eseguire. Per eseguire l'operazione TransactGetItems DynamoDB, il valore deve essere impostato su TransactGetItems. Questo valore è obbligatorio.

transactItems

Gli elementi di richiesta da includere. Il valore è un array di elementi di richiesta. Deve essere fornito almeno un elemento di richiesta. Questo valore transactItems è obbligatorio.

table

La tabella DynamoDB da cui recuperare l'elemento. Il valore è una stringa del nome della tabella. Questo valore table è obbligatorio.

key

La chiave DynamoDB che rappresenta la chiave primaria dell'elemento da recuperare. Gli elementi DynamoDB possono avere una sola chiave hash o una chiave hash e una chiave di ordinamento, a seconda della struttura della tabella. Per ulteriori informazioni su come specificare un «valore digitato», consulta [Sistema di tipi \(mappatura delle richieste\)](#).

projection

Una proiezione utilizzata per specificare gli attributi da restituire dall'operazione DynamoDB. [Per ulteriori informazioni sulle proiezioni, vedere Proiezioni](#). Questo campo è facoltativo.

Aspetti da ricordare:

- Se una transazione ha esito positivo, l'ordine degli elementi recuperati nel blocco items sarà lo stesso dell'ordine degli elementi della richiesta.
- Le transazioni vengono eseguite in qualche modo. all-or-nothing Se un elemento di richiesta causa un errore, l'intera transazione non viene eseguita e vengono restituiti i dettagli dell'errore.

- Un elemento di richiesta che non può essere recuperato non è un errore. Invece, un elemento null appare nel blocco elementi nella posizione corrispondente.
- Se l'errore di una transazione è `TransactionCanceledException`, il `cancellationReasons` blocco verrà popolato. L'ordine dei motivi di annullamento nel blocco `cancellationReasons` è lo stesso dell'ordine degli elementi della richiesta.
- `TransactGetItems` è limitato a 100 articoli richiesti.

Per il seguente esempio di modello di mappatura della richiesta:

```
{
  "version": "2018-05-29",
  "operation": "TransactGetItems",
  "transactItems": [
    ## First request item
    {
      "table": "posts",
      "key": {
        "post_id": {
          "S": "p1"
        }
      }
    },
    ## Second request item
    {
      "table": "authors",
      "key": {
        "author_id": {
          "S": "a1"
        }
      }
    }
  ]
}
```

Se la transazione ha esito positivo e viene recuperato solo il primo elemento richiesto, il risultato della chiamata disponibile `$ctx.result` è il seguente:

```
{
  "items": [
    {
      // Attributes of the first requested item
```

```

        "post_id": "p1",
        "post_title": "title",
        "post_description": "description"
    },
    // Could not retrieve the second requested item
    null,
],
"cancellationReasons": null
}

```

Se la transazione fallisce a `TransactionCanceledException` causa del primo elemento della richiesta, il risultato della chiamata disponibile in `$ctx.result` è il seguente:

```

{
  "items": null,
  "cancellationReasons": [
    {
      "type": "Sample error type",
      "message": "Sample error message"
    },
    {
      "type": "None",
      "message": "None"
    }
  ]
}

```

Il messaggio `$ctx.error` contiene dettagli relativi all'errore. La presenza delle chiavi `items` e `cancellationReasons` è garantita in `$ctx.result`.

Per un esempio più completo, segui il tutorial sulle transazioni di DynamoDB AppSync con questo [Tutorial: risolutori di transazioni DynamoDB](#).

TransactWriteItems

Il documento di mappatura delle `TransactWriteItems` richieste consente di indicare al AWS AppSync resolver DynamoDB di `TransactWriteItems` richiedere a DynamoDB di scrivere più elementi, potenzialmente su più tabelle. Per questo modello di richiesta, è necessario specificare quanto segue:

- Il nome della tabella di destinazione di ogni elemento di richiesta

- L'operazione di ogni elemento di richiesta da eseguire. Sono supportati quattro tipi di operazioni: `PutItem`, `UpdateItem`, `DeleteItem` e `ConditionCheck`
- La chiave di ogni elemento di richiesta da scrivere

Si applicano i limiti `TransactWriteItems` DynamoDB.

Il documento di mappatura `TransactWriteItems` ha la seguente struttura:

```
{
  "version": "2018-05-29",
  "operation": "TransactWriteItems",
  "transactItems": [
    {
      "table": "table1",
      "operation": "PutItem",
      "key": {
        "foo": ... typed value,
        "bar": ... typed value
      },
      "attributeValues": {
        "baz": ... typed value
      },
      "condition": {
        "expression": "someExpression",
        "expressionNames": {
          "#foo": "foo"
        },
        "expressionValues": {
          ":bar": ... typed value
        },
        "returnValuesOnConditionCheckFailure": true|false
      }
    },
    {
      "table": "table2",
      "operation": "UpdateItem",
      "key": {
        "foo": ... typed value,
        "bar": ... typed value
      },
      "update": {
        "expression": "someExpression",
```

```

        "expressionNames": {
            "#foo": "foo"
        },
        "expressionValues": {
            ":bar": ... typed value
        }
    },
    "condition": {
        "expression": "someExpression",
        "expressionNames": {
            "#foo": "foo"
        },
        "expressionValues": {
            ":bar": ... typed value
        },
        "returnValuesOnConditionCheckFailure": true|false
    }
},
{
    "table": "table3",
    "operation": "DeleteItem",
    "key": {
        "foo": ... typed value,
        "bar": ... typed value
    },
    "condition": {
        "expression": "someExpression",
        "expressionNames": {
            "#foo": "foo"
        },
        "expressionValues": {
            ":bar": ... typed value
        },
        "returnValuesOnConditionCheckFailure": true|false
    }
},
{
    "table": "table4",
    "operation": "ConditionCheck",
    "key": {
        "foo": ... typed value,
        "bar": ... typed value
    },
    "condition": {

```

```

        "expression": "someExpression",
        "expressionNames": {
            "#foo": "foo"
        },
        "expressionValues": {
            ":bar": ... typed value
        },
        "returnValuesOnConditionCheckFailure": true|false
    }
}
]
}

```

TransactWriteItems campi

TransactWriteItems elenco dei campi

I campi sono definiti come segue:

version

La versione di definizione del modello. Solo 2018-05-29 è supportata. Questo valore è obbligatorio.

operation

L'operazione DynamoDB da eseguire. Per eseguire l'operazione TransactWriteItems DynamoDB, il valore deve essere impostato su TransactWriteItems. Questo valore è obbligatorio.

transactItems

Gli elementi di richiesta da includere. Il valore è un array di elementi di richiesta. Deve essere fornito almeno un elemento di richiesta. Questo valore transactItems è obbligatorio.

Per PutItem, i campi sono definiti come segue:

table

La tabella DynamoDB di destinazione. Il valore è una stringa del nome della tabella. Questo valore table è obbligatorio.

operation

L'operazione DynamoDB da eseguire. Per eseguire l'operazione PutItem DynamoDB, il valore deve essere impostato su PutItem. Questo valore è obbligatorio.

key

La chiave DynamoDB che rappresenta la chiave primaria dell'elemento da inserire. Gli elementi DynamoDB possono avere una sola chiave hash o una chiave hash e una chiave di ordinamento, a seconda della struttura della tabella. Per ulteriori informazioni su come specificare un «valore digitato», consulta [Sistema di tipi \(mappatura delle richieste\)](#). Questo valore è obbligatorio.

attributeValues

Gli altri attributi della voce da inserir in DynamoDB. Per ulteriori informazioni su come specificare un «valore digitato», vedete [Type system \(request mapping\)](#). Questo campo è facoltativo.

condition

Una condizione per determinare se la richiesta deve riuscire o no in base allo stato dell'oggetto già incluso in DynamoDB. Se non viene specificata alcuna condizione, la richiesta PutItem sovrascrive qualsiasi valore esistente per quella voce. È possibile specificare se recuperare l'elemento esistente quando il controllo delle condizioni non riesce. [Per ulteriori informazioni sulle condizioni transazionali, consulta Espressioni delle condizioni di transazione.](#) Questo valore è facoltativo.

Per UpdateItem, i campi sono definiti come segue:

table

La tabella DynamoDB da aggiornare. Il valore è una stringa del nome della tabella. Questo valore table è obbligatorio.

operation

L'operazione DynamoDB da eseguire. Per eseguire l'operazione UpdateItem DynamoDB, il valore deve essere impostato su UpdateItem. Questo valore è obbligatorio.

key

La chiave DynamoDB che rappresenta la chiave primaria dell'elemento da aggiornare. Gli elementi DynamoDB possono avere una sola chiave hash o una chiave hash e una chiave di ordinamento, a seconda della struttura della tabella. Per ulteriori informazioni su come specificare un «valore digitato», consulta [Sistema di tipi \(mappatura delle richieste\)](#). Questo valore è obbligatorio.

update

La `update` sezione consente di specificare un'espressione di aggiornamento che descrive come aggiornare l'elemento in DynamoDB. Per ulteriori informazioni su come scrivere espressioni di aggiornamento, consulta la documentazione di [UpdateExpressions DynamoDB](#). Questa sezione è obbligatoria.

condition

Una condizione per determinare se la richiesta deve riuscire o no in base allo stato dell'oggetto già incluso in DynamoDB. Se non viene specificata alcuna condizione, la richiesta `UpdateItem` aggiorna qualsiasi valore esistente, indipendentemente dal suo stato attuale. È possibile specificare se recuperare l'elemento esistente quando il controllo delle condizioni non riesce. [Per ulteriori informazioni sulle condizioni transazionali, consulta Espressioni delle condizioni di transazione](#). Questo valore è facoltativo.

Per `DeleteItem`, i campi sono definiti come segue:

table

La tabella DynamoDB in cui eliminare l'elemento. Il valore è una stringa del nome della tabella. Questo valore `table` è obbligatorio.

operation

L'operazione DynamoDB da eseguire. Per eseguire l'operazione `DeleteItem` DynamoDB, il valore deve essere impostato su `DeleteItem`. Questo valore è obbligatorio.

key

La chiave DynamoDB che rappresenta la chiave primaria dell'elemento da eliminare. Gli elementi DynamoDB possono avere una sola chiave hash o una chiave hash e una chiave di ordinamento, a seconda della struttura della tabella. Per ulteriori informazioni su come specificare un «valore digitato», consulta [Sistema di tipi \(mappatura delle richieste\)](#). Questo valore è obbligatorio.

condition

Una condizione per determinare se la richiesta deve riuscire o no in base allo stato dell'oggetto già incluso in DynamoDB. Se non viene specificata alcuna condizione, la richiesta `DeleteItem` elimina la voce indipendentemente dal suo stato attuale. È possibile specificare se recuperare l'elemento esistente quando il controllo delle condizioni non

riesce. [Per ulteriori informazioni sulle condizioni transazionali, consulta Espressioni delle condizioni di transazione.](#) Questo valore è facoltativo.

Per ConditionCheck, i campi sono definiti come segue:

table

La tabella DynamoDB in cui controllare la condizione. Il valore è una stringa del nome della tabella. Questo valore `table` è obbligatorio.

operation

L'operazione DynamoDB da eseguire. Per eseguire l'operazione ConditionCheck DynamoDB, il valore deve essere impostato su ConditionCheck. Questo valore è obbligatorio.

key

La chiave DynamoDB che rappresenta la chiave primaria dell'elemento per il controllo delle condizioni. Gli elementi DynamoDB possono avere una sola chiave hash o una chiave hash e una chiave di ordinamento, a seconda della struttura della tabella. Per ulteriori informazioni su come specificare un «valore digitato», consulta [Sistema di tipi \(mappatura delle richieste\)](#). Questo valore è obbligatorio.

condition

Una condizione per determinare se la richiesta deve riuscire o no in base allo stato dell'oggetto già incluso in DynamoDB. È possibile specificare se recuperare l'elemento esistente quando il controllo delle condizioni non riesce. [Per ulteriori informazioni sulle condizioni transazionali, consulta Espressioni delle condizioni di transazione.](#) Questo valore è obbligatorio.

Aspetti da ricordare:

- Solo le chiavi degli elementi della richiesta vengono restituite nella risposta, in caso di esito positivo. L'ordine delle chiavi sarà lo stesso dell'ordine degli elementi della richiesta.
- Le transazioni vengono eseguite in qualsiasi modo. all-or-nothing Se un elemento di richiesta causa un errore, l'intera transazione non viene eseguita e vengono restituiti i dettagli dell'errore.
- Nessun elemento di richiesta può scegliere come target lo stesso elemento. Altrimenti causeranno TransactionCanceledExceptionerrori.

- Se l'errore di una transazione è `TransactionCanceledException`, il `cancellationReasons` blocco verrà popolato. Se il controllo delle condizioni di un elemento di richiesta fallisce e non è stato specificato `returnValuesOnConditionCheckFailure` come `false`, l'elemento esistente nella tabella viene recuperato e memorizzato in `item` nella posizione corrispondente del blocco `cancellationReasons`.
- `TransactWriteItems` è limitato a 100 articoli richiesti.
- Questa operazione non è supportata se utilizzata con il rilevamento dei conflitti. L'utilizzo di entrambi allo stesso tempo può causare un errore.

Per il seguente esempio di modello di mappatura della richiesta:

```
{
  "version": "2018-05-29",
  "operation": "TransactWriteItems",
  "transactItems": [
    {
      "table": "posts",
      "operation": "PutItem",
      "key": {
        "post_id": {
          "S": "p1"
        }
      },
      "attributeValues": {
        "post_title": {
          "S": "New title"
        },
        "post_description": {
          "S": "New description"
        }
      },
      "condition": {
        "expression": "post_title = :post_title",
        "expressionValues": {
          ":post_title": {
            "S": "Expected old title"
          }
        }
      }
    }
  ],
}
```

```

    "table": "authors",
    "operation": "UpdateItem",
    "key": {
      "author_id": {
        "S": "a1"
      },
    },
    "update": {
      "expression": "SET author_name = :author_name",
      "expressionValues": {
        ":author_name": {
          "S": "New name"
        }
      }
    },
  },
]
}

```

Se la transazione ha esito positivo, il risultato della chiamata disponibile in `$ctx.result` è il seguente:

```

{
  "keys": [
    // Key of the PutItem request
    {
      "post_id": "p1",
    },
    // Key of the UpdateItem request
    {
      "author_id": "a1"
    }
  ],
  "cancellationReasons": null
}

```

Se la transazione fallisce a causa del fallimento del controllo delle condizioni della PutItem richiesta, il risultato della chiamata disponibile in `$ctx.result` è il seguente:

```

{
  "keys": null,
  "cancellationReasons": [

```

```
{
  "item": {
    "post_id": "p1",
    "post_title": "Actual old title",
    "post_description": "Old description"
  },
  "type": "ConditionCheckFailed",
  "message": "The condition check failed."
},
{
  "type": "None",
  "message": "None"
}
]
```

Il messaggio `$ctx.error` contiene dettagli relativi all'errore. La presenza delle chiavi `keys` e `cancellationReasons` è garantita in `$ctx.result`.

Per un esempio più completo, segui il tutorial sulle transazioni di DynamoDB AppSync con questo [Tutorial: risolutori di transazioni DynamoDB](#).

Sistema di tipi (mappatura delle richieste)

Quando si utilizza il AWS AppSync resolver DynamoDB per chiamare le AWS AppSync tabelle DynamoDB, è necessario conoscere il tipo di ogni valore da utilizzare in quella chiamata. Questo perché DynamoDB supporta più primitive di tipo rispetto a GraphQL o JSON (come set e dati binari). AWS AppSync necessita di alcuni suggerimenti per la traduzione tra GraphQL e DynamoDB, altrimenti dovrebbe formulare alcune ipotesi su come i dati sono strutturati nella tabella.

[Per ulteriori informazioni sui tipi di dati DynamoDB, consulta i descrittori dei tipi di dati e la documentazione sui tipi di dati di DynamoDB.](#)

Un valore DynamoDB è rappresentato da un oggetto JSON contenente una singola coppia chiave-valore. La chiave specifica il tipo DynamoDB e il valore specifica il valore stesso. In questo esempio, la chiave `S` indica che il valore è una stringa e il valore `identifier` è il valore della stringa stessa.

```
{ "S" : "identifier" }
```

L'oggetto JSON non può contenere più di una coppia chiave-valore. In caso contrario, il documento di mappatura della richiesta non viene analizzato.

Un valore DynamoDB viene utilizzato ovunque in un documento di mappatura delle richieste in cui è necessario specificare un valore. ad esempio nelle sezioni `key` e `attributeValue`, nonché nella sezione `expressionValues` delle sezioni delle espressioni. Nell'esempio seguente, il `identifier` valore String di DynamoDB viene assegnato al campo in una sezione (magari in `key` `GetItem` un documento di mappatura `id` della richiesta).

```
"key" : {  
  "id" : { "S" : "identifier" }  
}
```

Tipi supportati

AWS AppSync supporta i seguenti tipi di scalari, documenti e set di DynamoDB:

S (tipo String)

Un singolo valore di stringa. Un valore di stringa DynamoDB è indicato da:

```
{ "S" : "some string" }
```

Un esempio di utilizzo è:

```
"key" : {  
  "id" : { "S" : "some string" }  
}
```

SS (tipo String set)

Un set di valori di stringa. Un valore DynamoDB String Set è indicato da:

```
{ "SS" : [ "first value", "second value", ... ] }
```

Un esempio di utilizzo è:

```
"attributeValues" : {  
  "phoneNumbers" : { "SS" : [ "+1 555 123 4567", "+1 555 234 5678" ] }  
}
```

N (tipo Number)

Un singolo valore numerico. Un valore numerico di DynamoDB è indicato da:

```
{ "N" : 1234 }
```

Un esempio di utilizzo è:

```
"expressionValues" : {  
  ":expectedVersion" : { "N" : 1 }  
}
```

NS (tipo Number set)

Un set di valori numerici. Un valore del DynamoDB Number Set è indicato da:

```
{ "NS" : [ 1, 2.3, 4 ... ] }
```

Un esempio di utilizzo è:

```
"attributeValues" : {  
  "sensorReadings" : { "NS" : [ 67.8, 12.2, 70 ] }  
}
```

B (tipo Binary)

Un valore binario. Un valore binario di DynamoDB è indicato da:

```
{ "B" : "SGVsbG8sIFdvcmxkIQo=" }
```

Nota che il valore è in realtà una stringa, dove la stringa è la rappresentazione codificata in base64 dei dati binari. AWS AppSync decodifica nuovamente questa stringa nel suo valore binario prima di inviarla a DynamoDB. AWS AppSync utilizza lo schema di decodifica base64 come definito da RFC 2045: qualsiasi carattere che non sia nell'alfabeto base64 viene ignorato.

Un esempio di utilizzo è:

```
"attributeValues" : {  
  "binaryMessage" : { "B" : "SGVsbG8sIFdvcmxkIQo=" }  
}
```

BS (tipo Binary set)

Un set di valori binari. Un valore del set binario di DynamoDB è indicato da:

```
{ "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ] }
```

Nota che il valore è in realtà una stringa, dove la stringa è la rappresentazione codificata in base64 dei dati binari. AWS AppSync decodifica nuovamente questa stringa nel suo valore binario prima di inviarla a DynamoDB. AWS AppSync utilizza lo schema di decodifica base64 come definito da RFC 2045: qualsiasi carattere che non sia nell'alfabeto base64 viene ignorato.

Un esempio di utilizzo è:

```
"attributeValues" : {  
  "binaryMessages" : { "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ] }  
}
```

BOOL (tipo Boolean)

Un valore booleano. Un valore booleano di DynamoDB è indicato da:

```
{ "BOOL" : true }
```

Solo i valori `true` e `false` sono validi.

Un esempio di utilizzo è:

```
"attributeValues" : {  
  "orderComplete" : { "BOOL" : false }  
}
```

L (tipo List)

Un elenco di qualsiasi altro valore DynamoDB supportato. Un valore di DynamoDB List è indicato da:

```
{ "L" : [ ... ] }
```

Nota che il valore è un valore composto, in cui l'elenco può contenere zero o più di qualsiasi valore DynamoDB supportato (inclusi altri elenchi). L'elenco può anche contenere una combinazione di diversi tipi.

Un esempio di utilizzo è:

```
{ "L" : [
```



```
{ "S" : "A string value" },  
  { "N" : 1 },  
  { "SS" : [ "Another string value", "Even more string values!" ] }  
}
```

M (tipo Map)

Rappresenta una raccolta non ordinata di coppie chiave-valore di altri valori DynamoDB supportati. Un valore di DynamoDB Map è indicato da:

```
{ "M" : { ... } }
```

Una mappa può contenere zero o più coppie chiave-valore. La chiave deve essere una stringa e il valore può essere qualsiasi valore DynamoDB supportato (incluse altre mappe). La mappa può anche contenere una combinazione di diversi tipi.

Un esempio di utilizzo è:

```
{ "M" : {  
    "someString" : { "S" : "A string value" },  
    "someNumber" : { "N" : 1 },  
    "stringSet" : { "SS" : [ "Another string value", "Even more string  
values!" ] }  
  }  
}
```

NULL (tipo Null)

Un valore nullo. Un valore Null di DynamoDB è indicato da:

```
{ "NULL" : null }
```

Un esempio di utilizzo è:

```
"attributeValues" : {  
  "phoneNumbers" : { "NULL" : null }  
}
```

Per ulteriori informazioni su ciascun tipo, consulta la [documentazione di DynamoDB](#).

Sistema di tipi (mappatura delle risposte)

Quando riceve una risposta da DynamoDB AWS AppSync , la converte automaticamente in tipi primitivi GraphQL e JSON. Ogni attributo in DynamoDB viene decodificato e restituito nel contesto di mappatura delle risposte.

Ad esempio, se DynamoDB restituisce quanto segue:

```
{
  "id" : { "S" : "1234" },
  "name" : { "S" : "Nadia" },
  "age" : { "N" : 25 }
}
```

Quindi il resolver AWS AppSync DynamoDB lo converte nei tipi GraphQL e JSON come:

```
{
  "id" : "1234",
  "name" : "Nadia",
  "age" : 25
}
```

Questa sezione spiega come AWS AppSync convertire i seguenti tipi di scalari, documenti e set di DynamoDB:

S (tipo String)

Un singolo valore di stringa. Un valore DynamoDB String viene restituito come stringa.

Ad esempio, se DynamoDB ha restituito il seguente valore DynamoDB String:

```
{ "S" : "some string" }
```

AWS AppSync lo converte in una stringa:

```
"some string"
```

SS (tipo String set)

Un set di valori di stringa. Un valore DynamoDB String Set viene restituito come elenco di stringhe.

Ad esempio, se DynamoDB ha restituito il seguente valore DynamoDB String Set:

```
{ "SS" : [ "first value", "second value", ... ] }
```

AWS AppSync lo converte in un elenco di stringhe:

```
[ "+1 555 123 4567", "+1 555 234 5678" ]
```

N (tipo Number)

Un singolo valore numerico. Un valore numerico di DynamoDB viene restituito come numero.

Ad esempio, se DynamoDB ha restituito il seguente valore numerico di DynamoDB:

```
{ "N" : 1234 }
```

AWS AppSync lo converte in un numero:

```
1234
```

NS (tipo Number set)

Un set di valori numerici. Un valore del set di numeri di DynamoDB viene restituito come elenco di numeri.

Ad esempio, se DynamoDB ha restituito il seguente valore del DynamoDB Number Set:

```
{ "NS" : [ 67.8, 12.2, 70 ] }
```

AWS AppSync lo converte in un elenco di numeri:

```
[ 67.8, 12.2, 70 ]
```

B (tipo Binary)

Un valore binario. Un valore binario di DynamoDB viene restituito come stringa contenente la rappresentazione in base64 di quel valore.

Ad esempio, se DynamoDB ha restituito il seguente valore binario di DynamoDB:

```
{ "B" : "SGVsbG8sIFdvcmxkIQo=" }
```

AWS AppSync lo converte in una stringa contenente la rappresentazione in base64 del valore:

```
"SGVsbG8sIFdvcmxkIQo="
```

I dati binari vengono codificati nello schema di codifica base64 secondo quanto specificato negli standard [RFC 4648](#) e [RFC 2045](#).

BS (tipo Binary set)

Un set di valori binari. Un valore del set binario di DynamoDB viene restituito come elenco di stringhe contenenti la rappresentazione in base64 dei valori.

Ad esempio, se DynamoDB ha restituito il seguente valore di DynamoDB Binary Set:

```
{ "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ] }
```

AWS AppSync lo converte in un elenco di stringhe contenenti la rappresentazione in base64 dei valori:

```
[ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ]
```

I dati binari vengono codificati nello schema di codifica base64 secondo quanto specificato negli standard [RFC 4648](#) e [RFC 2045](#).

BOOL (tipo Boolean)

Un valore booleano. Un valore booleano DynamoDB viene restituito come booleano.

Ad esempio, se DynamoDB ha restituito il seguente valore booleano di DynamoDB:

```
{ "BOOL" : true }
```

AWS AppSync lo converte in booleano:

```
true
```

L (tipo List)

Un elenco di qualsiasi altro valore DynamoDB supportato. Un valore DynamoDB List viene restituito come elenco di valori, in cui viene convertito anche ogni valore interno.

Ad esempio, se DynamoDB ha restituito il seguente valore di DynamoDB List:

```
{ "L" : [  
  { "S" : "A string value" },  
  { "N" : 1 },  
  { "SS" : [ "Another string value", "Even more string values!" ] }  
]
```

AWS AppSync lo converte in un elenco di valori convertiti:

```
[ "A string value", 1, [ "Another string value", "Even more string values!" ] ]
```

M (tipo Map)

Una key/value raccolta di qualsiasi altro valore DynamoDB supportato. Un valore della mappa DynamoDB viene restituito come oggetto JSON, dove key/value ogni valore viene anche convertito.

Ad esempio, se DynamoDB ha restituito il seguente valore di DynamoDB Map:

```
{ "M" : {  
  "someString" : { "S" : "A string value" },  
  "someNumber" : { "N" : 1 },  
  "stringSet" : { "SS" : [ "Another string value", "Even more string  
values!" ] }  
}
```

AWS AppSync lo converte in un oggetto JSON:

```
{  
  "someString" : "A string value",  
  "someNumber" : 1,  
  "stringSet" : [ "Another string value", "Even more string values!" ]  
}
```

NULL (tipo Null)

Un valore nullo.

Ad esempio, se DynamoDB ha restituito il seguente valore Null di DynamoDB:

```
{ "NULL" : null }
```

AWS AppSync lo converte in un valore nullo:

```
null
```

Filtri

Quando si eseguono interrogazioni su oggetti in DynamoDB utilizzando Query le operazioni Scan and, è possibile facoltativamente specificare `filter` un valore che valuti i risultati e restituisca solo i valori desiderati.

La sezione di mappatura dei filtri di un documento di mappatura di Query o Scan ha la seguente struttura:

```
"filter" : {  
  "expression" : "filter expression"  
  "expressionNames" : {  
    "#name" : "name",  
  },  
  "expressionValues" : {  
    ":value" : ... typed value  
  },  
}
```

I campi sono definiti come segue:

expression

L'espressione della query. Per ulteriori informazioni su come scrivere espressioni di filtro, consulta la documentazione di [DynamoDB e QueryFilter ScanFilter](#) DynamoDB. Questo campo deve essere specificato.

expressionNames

Le sostituzioni per i segnaposto dell'attributo di espressione name sotto forma di coppie chiave-valore. La chiave corrisponde a un segnaposto nome utilizzato in `expression`. Il valore deve essere una stringa che corrisponde al nome dell'attributo dell'elemento in DynamoDB. Questo è un campo facoltativo in cui vanno riportate solo le sostituzioni per i segnaposto dell'attributo di espressione name utilizzate in `expression`.

expressionValues

Le sostituzioni per i segnaposto dell'attributo di espressione value sotto forma di coppie chiave-valore. La chiave corrisponde a un segnaposto per un valore utilizzato in `expression`, mentre il valore deve essere un valore tipizzato. Per ulteriori informazioni su come specificare un "valore tipizzato", consulta [Sistema di tipi \(mappatura della richiesta\)](#). Questo elemento deve essere specificato. Questo è un campo facoltativo in cui vanno riportate solo le sostituzioni per i segnaposto dell'attributo di espressione value utilizzate in `expression`.

Esempio

L'esempio seguente è una sezione filtro per un modello di mappatura, in cui le voci recuperate da DynamoDB vengono restituite solo se il titolo inizia con l'argomento. `title`

```
"filter" : {
  "expression" : "begins_with(#title, :title)",
  "expressionNames" : {
    "#title" : "title"
  },
  "expressionValues" : {
    ":title" : $util.dynamodb.toDynamoDBJson($context.arguments.title)
  }
}
```

Espressioni di condizione

Quando modificate gli oggetti in DynamoDB utilizzando le operazioni `PutItem`, e `DeleteItem` `DynamoDBUpdateItem`, potete facoltativamente specificare un'espressione di condizione che controlli se la richiesta deve avere successo o meno, in base allo stato dell'oggetto già in DynamoDB prima dell'esecuzione dell'operazione.

Il AWS AppSync resolver DynamoDB consente di specificare `PutItem` un'espressione di condizione `DeleteItem` e richiedere documenti di mappatura `UpdateItem`, nonché una strategia da seguire se la condizione fallisce e l'oggetto non è stato aggiornato.

Esempio 1

Il seguente documento di mappatura `PutItem` non contiene un'espressione di condizione. Di conseguenza, inserisce un elemento in DynamoDB anche se esiste già un elemento con la stessa chiave, sovrascrivendo così l'elemento esistente.

```
{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key" : {
    "id" : { "S" : "1" }
  }
}
```

Esempio 2

Il seguente documento di PutItem mappatura contiene un'espressione di condizione che consente il successo dell'operazione solo se un elemento con la stessa chiave non esiste in DynamoDB.

```
{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key" : {
    "id" : { "S" : "1" }
  },
  "condition" : {
    "expression" : "attribute_not_exists(id)"
  }
}
```

Per impostazione predefinita, se il controllo delle condizioni fallisce, il resolver AWS AppSync DynamoDB restituisce un errore per la mutazione. Tuttavia, il AWS AppSync resolver DynamoDB offre alcune funzionalità aggiuntive per aiutare gli sviluppatori a gestire alcuni casi limite comuni:

- Se AWS AppSync il resolver DynamoDB è in grado di determinare che il valore corrente in DynamoDB corrisponde al risultato desiderato, considera l'operazione come se fosse riuscita comunque.
- Invece di restituire un errore, puoi configurare il resolver per richiamare una funzione Lambda personalizzata per decidere come il resolver DynamoDB deve gestire AWS AppSync l'errore.

Queste vengono descritte in dettaglio nella sezione [Gestione di un errore nel controllo della condizione](#).

[Per ulteriori informazioni sulle espressioni delle condizioni di DynamoDB, consulta la documentazione di DynamoDB. ConditionExpressions](#)

Specificare una condizione

I documenti di mappatura della richiesta `PutItem`, `UpdateItem` e `DeleteItem` consentono tutti di specificare una sezione facoltativa `condition`. Se omessa, non vengono eseguiti controlli di condizione. Se specificata, la condizione deve essere soddisfatta perché l'operazione abbia esito positivo.

Una sezione `condition` ha la seguente struttura:

```
"condition" : {  
  "expression" : "someExpression"  
  "expressionNames" : {  
    "#foo" : "foo"  
  },  
  "expressionValues" : {  
    ":bar" : ... typed value  
  },  
  "equalsIgnore" : [ "version" ],  
  "consistentRead" : true,  
  "conditionalCheckFailedHandler" : {  
    "strategy" : "Custom",  
    "lambdaArn" : "arn:..."  
  }  
}
```

I seguenti campi specificano la condizione:

expression

L'espressione di aggiornamento in sé. Per ulteriori informazioni su come scrivere espressioni di condizione, consulta la documentazione di [ConditionExpressions DynamoDB](#). Questo campo deve essere specificato.

expressionNames

Le sostituzioni per i segnaposto dell'attributo di espressione name, sotto forma di coppie chiave-valore. La chiave corrisponde a un segnaposto per il nome utilizzato nell'espressione e il valore deve essere una stringa corrispondente al nome dell'attributo dell'elemento in DynamoDB. Questo è un campo facoltativo in cui vanno riportate solo le sostituzioni per i segnaposto dell'attributo di espressione name utilizzate in `expression`.

expressionValues

Le sostituzioni per i segnaposto del valore dell'attributo di espressione, sotto forma di coppie chiave-valore. La chiave corrisponde a un segnaposto per un valore utilizzato nell'espressione, mentre il valore deve essere un valore tipizzato. Per ulteriori informazioni su come specificare un "valore tipizzato", consulta [Sistema di tipi \(mappatura della richiesta\)](#). Questo elemento deve essere specificato. Questo è un campo facoltativo in cui vanno riportate solo le sostituzioni per i segnaposto dell'attributo di espressione value utilizzate nell'espressione.

I campi rimanenti indicano al resolver AWS AppSync DynamoDB come gestire un errore di controllo delle condizioni:

equalsIgnore

Quando un controllo delle condizioni fallisce durante l'utilizzo dell'PutItemoperazione, il resolver AWS AppSync DynamoDB confronta l'elemento attualmente in DynamoDB con l'elemento che ha cercato di scrivere. Se sono uguali, tratta l'operazione come se avesse avuto comunque esito positivo. È possibile utilizzare il equalsIgnore campo per specificare un elenco di attributi da ignorare quando si esegue il AWS AppSync confronto. Ad esempio, se l'unica differenza era un version attributo, considera l'operazione come se fosse riuscita. Questo campo è facoltativo.

consistentRead

Quando un controllo delle condizioni fallisce, AWS AppSync ottiene il valore corrente dell'elemento da DynamoDB utilizzando una lettura fortemente coerente. È possibile utilizzare questo campo per indicare al resolver AWS AppSync DynamoDB di utilizzare invece una lettura alla fine coerente. Si tratta di un campo facoltativo, impostato di default su true.

conditionalCheckFailedHandler

Questa sezione consente di specificare in che modo il resolver AWS AppSync DynamoDB tratta un errore di controllo delle condizioni dopo aver confrontato il valore corrente in DynamoDB con il risultato previsto. Questa sezione è facoltativa. Se omissa, la strategia predefinita è Reject.

strategy

La strategia adottata dal AWS AppSync resolver DynamoDB dopo aver confrontato il valore corrente in DynamoDB con il risultato previsto. Questo campo è obbligatorio e ha i seguenti possibili valori:

Reject

La mutazione fallisce e viene aggiunto un errore alla risposta GraphQL.

Custom

Il AWS AppSync resolver DynamoDB richiama una funzione Lambda personalizzata per decidere come gestire l'errore del controllo delle condizioni. Se `strategy` è impostato su `Custom`, il campo `lambdaArn` deve contenere l'ARN della funzione Lambda da invocare.

lambdaArn

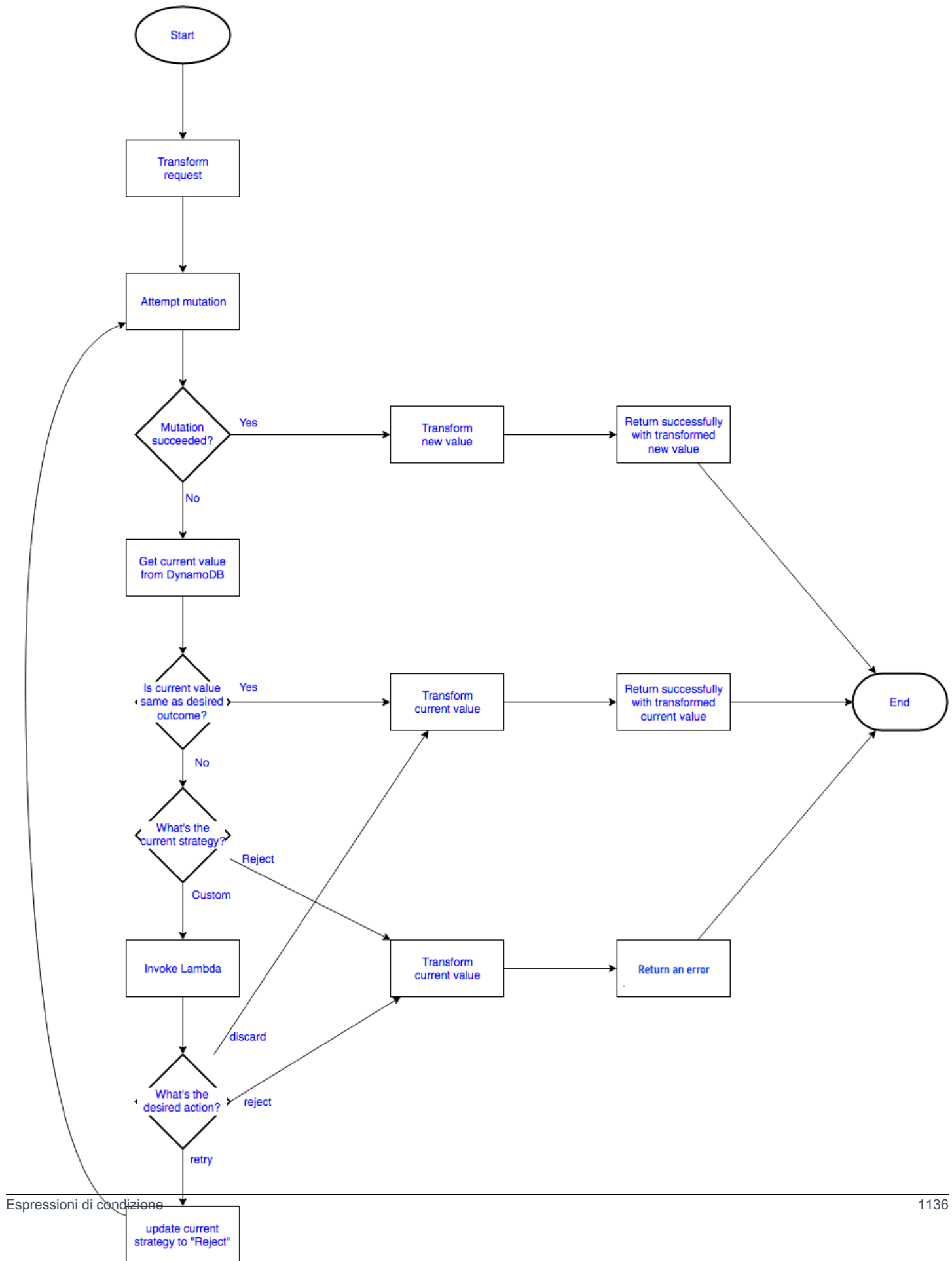
L'ARN della funzione Lambda da richiamare che determina in che modo il resolver DynamoDB deve gestire l'errore del controllo delle AWS AppSync condizioni. Questo campo deve essere specificato solo se `strategy` è impostato su `Custom`. Per ulteriori informazioni su come utilizzare questa funzionalità, consulta [Gestione di un errore nel controllo della condizione](#).

Gestione di un errore di controllo delle condizioni

Per impostazione predefinita, quando un controllo delle condizioni fallisce, il resolver AWS AppSync DynamoDB restituisce un errore per la mutazione e il valore corrente dell'oggetto in DynamoDB. Tuttavia, il AWS AppSync resolver DynamoDB offre alcune funzionalità aggiuntive per aiutare gli sviluppatori a gestire alcuni casi limite comuni:

- Se AWS AppSync il resolver DynamoDB è in grado di determinare che il valore corrente in DynamoDB corrisponde al risultato desiderato, considera l'operazione come se fosse riuscita comunque.
- Invece di restituire un errore, puoi configurare il resolver per richiamare una funzione Lambda personalizzata per decidere come il resolver DynamoDB deve gestire AWS AppSync l'errore.

Il diagramma di questo processo è il seguente:



Verifica del risultato desiderato

Quando il controllo delle condizioni fallisce, il resolver AWS AppSync DynamoDB esegue `GetItem` una richiesta DynamoDB per ottenere il valore corrente dell'elemento da DynamoDB. Per impostazione predefinita, si utilizza una lettura fortemente consistente, ma è possibile intervenire sulla configurazione utilizzando il campo `consistentRead` del blocco `condition` e confrontandolo con il risultato previsto:

- Per l'operazione `PutItem`, il resolver AWS AppSync DynamoDB confronta il valore corrente con quello che ha tentato di scrivere, escludendo dal confronto tutti gli attributi elencati. `equalsIgnore` Se gli elementi sono uguali, considera l'operazione come riuscita e restituisce l'elemento recuperato da DynamoDB. In caso contrario, viene seguita la strategia configurata.

Ad esempio, se il documento di mappatura della richiesta `PutItem` fosse simile al seguente:

```
{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key" : {
    "id" : { "S" : "1" }
  },
  "attributeValues" : {
    "name" : { "S" : "Steve" },
    "version" : { "N" : 2 }
  },
  "condition" : {
    "expression" : "version = :expectedVersion",
    "expressionValues" : {
      ":expectedVersion" : { "N" : 1 }
    },
    "equalsIgnore": [ "version" ]
  }
}
```

E la voce attualmente inclusa in DynamoDB fosse simile alla seguente:

```
{
  "id" : { "S" : "1" },
  "name" : { "S" : "Steve" },
  "version" : { "N" : 8 }
}
```

Il AWS AppSync resolver DynamoDB confronta l'elemento che ha cercato di scrivere con il valore corrente, verifica che l'unica differenza è `version` il campo, ma poiché è configurato per ignorare il campo, considera `version` l'operazione come riuscita e restituisce l'elemento recuperato da DynamoDB.

- Per l'`DeleteItem` operazione, il resolver AWS AppSync DynamoDB verifica che un elemento sia stato restituito da DynamoDB. Se nessuna voce è stata restituita, considera l'operazione riuscita. In caso contrario, viene seguita la strategia configurata.
- Per l'`UpdateItem` operazione, il AWS AppSync resolver DynamoDB non dispone di informazioni sufficienti per determinare se l'elemento attualmente in DynamoDB corrisponde al risultato previsto e pertanto segue la strategia configurata.

Se lo stato corrente dell'oggetto in DynamoDB è diverso dal risultato previsto, il resolver AWS AppSync DynamoDB segue la strategia configurata, rifiutando la mutazione o richiamando una funzione Lambda per determinare cosa fare dopo.

Seguendo la strategia di «rifiuto»

Quando si segue la `Reject` strategia, il resolver AWS AppSync DynamoDB restituisce un errore per la mutazione.

Ad esempio, partendo dalla richiesta di mutazione seguente:

```
mutation {  
  updatePerson(id: 1, name: "Steve", expectedVersion: 1) {  
    Name  
    theVersion  
  }  
}
```

Se l'elemento restituito da DynamoDB è simile al seguente:

```
{  
  "id" : { "S" : "1" },  
  "name" : { "S" : "Steve" },  
  "version" : { "N" : 8 }  
}
```

E il modello di mappatura della risposta fosse simile al seguente:

```
{
  "id" : $util.toJson($context.result.id),
  "Name" : $util.toJson($context.result.name),
  "theVersion" : $util.toJson($context.result.version)
}
```

La risposta GraphQL ha il seguente aspetto:

```
{
  "data": null,
  "errors": [
    {
      "message": "The conditional request failed (Service: AmazonDynamoDBv2;
Status Code: 400; Error Code: ConditionalCheckFailedException; Request ID:
ABCDEFGHJKLMNOPQRSTUVWXYZABCDEFGHJKLMNOPQRSTUVWXYZ)"
      "errorType": "DynamoDB:ConditionalCheckFailedException",
      ...
    }
  ]
}
```

Inoltre, se i campi dell'oggetto restituito sono compilati da altri resolver in caso di successo della mutazione, tali campi non saranno risolti quando l'oggetto viene restituito nella sezione `error`.

Seguendo la strategia «personalizzata»

Quando segue la Custom strategia, il resolver AWS AppSync DynamoDB richiama una funzione Lambda per decidere cosa fare dopo. La funzione Lambda sceglie una delle seguenti opzioni:

- Rifiuto della mutazione (`reject`). Ciò indica al AWS AppSync resolver DynamoDB di comportarsi come se lo `Reject` fosse la strategia configurata, restituendo un errore per la mutazione e il valore corrente dell'oggetto in DynamoDB come descritto nella sezione precedente.
- Rifiuto della mutazione (`discard`). Ciò indica al resolver AWS AppSync DynamoDB di ignorare silenziosamente l'errore del controllo delle condizioni e restituisce il valore in DynamoDB.
- Rifiuto della mutazione (`retry`). Questo indica al resolver AWS AppSync DynamoDB di riprovare la mutazione con un nuovo documento di mappatura delle richieste.

La richiesta di invocazione Lambda

Il AWS AppSync resolver DynamoDB richiama la funzione Lambda specificata in `lambdaArn`. Utilizza lo stesso `service-role-arn` configurato per l'origine dati. Il payload dell'invocazione ha la seguente struttura:

```
{
  "arguments": { ... },
  "requestMapping": {... },
  "currentValue": { ... },
  "resolver": { ... },
  "identity": { ... }
}
```

I campi sono definiti come segue:

arguments

Gli argomenti della mutazione GraphQL. Sono uguali agli argomenti disponibili per il documento di mappatura della richiesta in `$context.arguments`.

requestMapping

Il documento di mappatura della richiesta per questa operazione.

currentValue

Il valore corrente dell'oggetto in DynamoDB.

resolver

Informazioni sul resolver. AWS AppSync

identity

Informazioni sul chiamante. Sono uguali alle informazioni sull'identità disponibili per il documento di mappatura della richiesta in `$context.identity`.

Un esempio completo di payload:

```
{
  "arguments": {
    "id": "1",
    "name": "Steve",
    "expectedVersion": 1
  },

```



```

"requestMapping": {
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key" : {
    "id" : { "S" : "1" }
  },
  "attributeValues" : {
    "name" : { "S" : "Steve" },
    "version" : { "N" : 2 }
  },
  "condition" : {
    "expression" : "version = :expectedVersion",
    "expressionValues" : {
      ":expectedVersion" : { "N" : 1 }
    },
    "equalsIgnore": [ "version" ]
  }
},
"currentValue": {
  "id" : { "S" : "1" },
  "name" : { "S" : "Steve" },
  "version" : { "N" : 8 }
},
"resolver": {
  "tableName": "People",
  "awsRegion": "us-west-2",
  "parentType": "Mutation",
  "field": "updatePerson",
  "outputType": "Person"
},
"identity": {
  "accountId": "123456789012",
  "sourceIp": "x.x.x.x",
  "user": "AIDAAAAAAAAAAAAAAAAAAAA",
  "userArn": "arn:aws:iam::123456789012:user/appsync"
}
}

```

La risposta all'invocazione Lambda

La funzione Lambda può ispezionare il payload di chiamata e applicare qualsiasi logica aziendale per decidere come il resolver DynamoDB deve gestire l'errore AWS AppSync . Sono disponibili tre opzioni per la gestione dell'errore nel controllo della condizione:

- Rifiuto della mutazione (`reject`). Il payload di risposta per questa opzione deve avere questa struttura:

```
{
  "action": "reject"
}
```

Ciò indica al AWS AppSync resolver DynamoDB di comportarsi come se la strategia configurata lo `Reject` fosse, restituendo un errore per la mutazione e il valore corrente dell'oggetto in DynamoDB, come descritto nella sezione precedente.

- Rifiuto della mutazione (`discard`). Il payload di risposta per questa opzione deve avere questa struttura:

```
{
  "action": "discard"
}
```

Ciò indica al resolver AWS AppSync DynamoDB di ignorare silenziosamente l'errore del controllo delle condizioni e restituisce il valore in DynamoDB.

- Rifiuto della mutazione (`retry`). Il payload di risposta per questa opzione deve avere questa struttura:

```
{
  "action": "retry",
  "retryMapping": { ... }
}
```

Questo indica al resolver AWS AppSync DynamoDB di riprovare la mutazione con un nuovo documento di mappatura delle richieste. La struttura della `retryMapping` sezione dipende dall'operazione DynamoDB ed è un sottoinsieme del documento completo di mappatura delle richieste per quell'operazione.

Per `PutItem`, la sezione `retryMapping` ha la seguente struttura. Per una descrizione del campo, vedere `attributeValues` [PutItem](#)

```
{
  "attributeValues": { ... },
  "condition": {
```

```

    "equalsIgnore" = [ ... ],
    "consistentRead" = true
  }
}
```

Per `UpdateItem`, la sezione `retryMapping` ha la seguente struttura. Per una descrizione della `update` sezione, vedere [UpdateItem](#).

```

{
  "update" : {
    "expression" : "someExpression"
    "expressionNames" : {
      "#foo" : "foo"
    },
    "expressionValues" : {
      ":bar" : ... typed value
    }
  },
  "condition": {
    "consistentRead" = true
  }
}
```

Per `DeleteItem`, la sezione `retryMapping` ha la seguente struttura.

```

{
  "condition": {
    "consistentRead" = true
  }
}
```

Non c'è modo di specificare operazioni o chiavi diverse su cui lavorare. Il AWS AppSync resolver DynamoDB consente solo di ripetere la stessa operazione sullo stesso oggetto. Inoltre, la sezione `condition` non consente di specificare un valore per `conditionalCheckFailedHandler`. Se il nuovo tentativo fallisce, il resolver AWS AppSync DynamoDB segue la strategia. `Reject`

Ecco un esempio di funzione Lambda per far fronte a una richiesta `PutItem` non riuscita. La logica di business osserva l'autore della chiamata. Se è stata effettuata da `jeffTheAdmin`, riprova la richiesta,

aggiornando l'`version` e `expectedVersion` dall'elemento attualmente in DynamoDB. Altrimenti rifiuta la mutazione.

```
exports.handler = (event, context, callback) => {
  console.log("Event: " + JSON.stringify(event));

  // Business logic goes here.

  var response;
  if ( event.identity.user == "jeffTheAdmin" ) {
    response = {
      "action" : "retry",
      "retryMapping" : {
        "attributeValues" : event.requestMapping.attributeValues,
        "condition" : {
          "expression" : event.requestMapping.condition.expression,
          "expressionValues" :
event.requestMapping.condition.expressionValues
        }
      }
    }
    response.retryMapping.attributeValues.version = { "N" :
event.currentValue.version.N + 1 }
    response.retryMapping.condition.expressionValues[':expectedVersion'] =
event.currentValue.version

  } else {
    response = { "action" : "reject" }
  }

  console.log("Response: " + JSON.stringify(response))
  callback(null, response)
};
```

Espressioni relative alle condizioni delle transazioni

Le espressioni di condizione delle transazioni sono disponibili nei modelli di mappatura delle richieste di tutti e quattro i tipi di operazioni in `TransactWriteItems`, vale a dire `PutItem`, `DeleteItem`, `UpdateItem` e `ConditionCheck`.

Per `PutItem`, e `DeleteItem` `UpdateItem`, l'espressione della condizione di transazione è facoltativa. Per `ConditionCheck`, l'espressione della condizione della transazione è obbligatoria.

Esempio 1

Il seguente documento di mappatura DeleteItem transazionale non dispone di un'espressione di condizione. Di conseguenza, elimina l'elemento in DynamoDB.

```
{
  "version": "2018-05-29",
  "operation": "TransactWriteItems",
  "transactItems": [
    {
      "table": "posts",
      "operation": "DeleteItem",
      "key": {
        "id": { "S" : "1" }
      }
    }
  ]
}
```

Esempio 2

Il seguente documento di DeleteItem mappatura transazionale contiene un'espressione della condizione di transazione che consente l'esito positivo dell'operazione solo se l'autore di quel post è uguale a un determinato nome.

```
{
  "version": "2018-05-29",
  "operation": "TransactWriteItems",
  "transactItems": [
    {
      "table": "posts",
      "operation": "DeleteItem",
      "key": {
        "id": { "S" : "1" }
      },
      "condition": {
        "expression": "author = :author",
        "expressionValues": {
          ":author": { "S" : "Chunyan" }
        }
      }
    }
  ]
}
```

```
]
}
```

Se il controllo della condizione fallisce, genera una `TransactionCanceledException` e il dettaglio dell'errore viene restituito in `$ctx.result.cancellationReasons`. Tieni presente che, per impostazione predefinita, verrà restituito il vecchio elemento in DynamoDB che ha impedito il controllo delle condizioni. `$ctx.result.cancellationReasons`

Specificare una condizione

I documenti di mappatura della richiesta `PutItem`, `UpdateItem` e `DeleteItem` consentono tutti di specificare una sezione facoltativa `condition`. Se omessa, non vengono eseguiti controlli di condizione. Se specificata, la condizione deve essere soddisfatta perché l'operazione abbia esito positivo. Per `ConditionCheck` deve essere specificata una sezione `condition`. La condizione deve essere vera affinché l'intera transazione abbia esito positivo.

Una sezione `condition` ha la seguente struttura:

```
"condition": {
  "expression": "someExpression",
  "expressionNames": {
    "#foo": "foo"
  },
  "expressionValues": {
    ":bar": ... typed value
  },
  "returnValuesOnConditionCheckFailure": false
}
```

I seguenti campi specificano la condizione:

expression

L'espressione di aggiornamento in sé. Per ulteriori informazioni su come scrivere espressioni di condizione, consulta la documentazione di [ConditionExpressions DynamoDB](#). Questo campo deve essere specificato.

expressionNames

Le sostituzioni per i segnaposto dell'attributo di espressione name, sotto forma di coppie chiave-valore. La chiave corrisponde a un segnaposto per il nome utilizzato nell'espressione e il valore deve essere una stringa corrispondente al nome dell'attributo dell'elemento in DynamoDB. Questo

è un campo facoltativo in cui vanno riportate solo le sostituzioni per i segnaposto dell'attributo di espressione name utilizzate inexpression.

expressionValues

Le sostituzioni per i segnaposto del valore dell'attributo di espressione, sotto forma di coppie chiave-valore. La chiave corrisponde a un segnaposto per un valore utilizzato nell'espressione, mentre il valore deve essere un valore tipizzato. Per ulteriori informazioni su come specificare un "valore tipizzato", consulta Sistema di tipi (mappatura della richiesta). Questo elemento deve essere specificato. Questo è un campo facoltativo in cui vanno riportate solo le sostituzioni per i segnaposto dell'attributo di espressione value utilizzate nell'espressione.

returnValuesOnConditionCheckFailure

Specificare se recuperare l'elemento in DynamoDB quando un controllo delle condizioni fallisce. L'elemento recuperato sarà in `$ctx.result.cancellationReasons[$index].item`, dove `$index` è l'indice dell'elemento richiesta che non ha superato il controllo della condizione. Il valore predefinito di questo valore è true.

Proiezioni

Quando si leggono oggetti in DynamoDB utilizzando `GetItem` le operazioni `Scan`, `QueryBatchGetItem`, `TransactGetItems` e,, è possibile specificare facoltativamente una proiezione che identifichi gli attributi desiderati. La proiezione ha la seguente struttura, simile ai filtri:

```
"projection" : {
  "expression" : "projection expression"
  "expressionNames" : {
    "#name" : "name",
  }
}
```

I campi sono definiti come segue:

expression

L'espressione di proiezione, che è una stringa. Per recuperare un singolo attributo, specificane il nome. Per più attributi, i nomi devono essere valori separati da virgole. Per ulteriori informazioni sulla scrittura di espressioni di proiezione, consulta la documentazione delle espressioni di proiezione [DynamoDB](#). Questo campo è obbligatorio.

expressionNames

Le sostituzioni degli attributi di espressione chiamano segnaposto sotto forma di coppie chiave-valore. La chiave corrisponde a un segnaposto nome utilizzato in `expression`. Il valore deve essere una stringa che corrisponde al nome dell'attributo dell'elemento in DynamoDB. Questo campo è facoltativo e deve essere compilato solo con sostituzioni per i segnaposto dei nomi degli attributi di espressione utilizzati in `expression`. Per ulteriori informazioni su `expressionNames`, consulta la documentazione di [DynamoDB](#).

Esempio 1

L'esempio seguente è una sezione di proiezione per un modello di mappatura VTL in cui `id` vengono restituiti solo gli attributi `author` e da DynamoDB:

```
"projection" : {  
  "expression" : "#author, id",  
  "expressionNames" : {  
    "#author" : "author"  
  }  
}
```

Tip

[È possibile accedere al set di selezione delle richieste GraphQL utilizzando `\$context.info.selectionSetList`](#). Questo campo consente di inquadrare l'espressione di proiezione in modo dinamico in base alle proprie esigenze.

Note

Quando si utilizzano espressioni di proiezione con le Scan operazioni Query and, il valore per `select` deve essere `SPECIFIC_ATTRIBUTES`. Per ulteriori informazioni, consulta la documentazione di [DynamoDB](#).

AWS AppSync riferimento al modello di mappatura del resolver per RDS

I modelli di mappatura AWS AppSync dei resolver RDS consentono agli sviluppatori di inviare query SQL a un'API di dati per Amazon Aurora Serverless e ottenere il risultato di queste query.

Richiedi un modello di mappatura

Il modello di mappatura della richiesta per RDS è abbastanza semplice:

```
{
  "version": "2018-05-29",
  "statements": [],
  "variableMap": {},
  "variableTypeHintMap": {}
}
```

Di seguito è riportata la rappresentazione dello schema JSON del modello di mappatura della richiesta RDS, dopo la risoluzione.

```
{
  "definitions": {},
  "$schema": "https://json-schema.org/draft-07/schema#",
  "$id": "https://example.com/root.json",
  "type": "object",
  "title": "The Root Schema",
  "required": [
    "version",
    "statements",
    "variableMap"
  ],
  "properties": {
    "version": {
      "$id": "#/properties/version",
      "type": "string",
      "title": "The Version Schema",
      "default": "",
      "examples": [
        "2018-05-29"
      ],
      "enum": [
```

```

        "2018-05-29"
      ],
      "pattern": "^(.*)$"
    },
    "statements": {
      "$id": "#/properties/statements",
      "type": "array",
      "title": "The Statements Schema",
      "items": {
        "$id": "#/properties/statements/items",
        "type": "string",
        "title": "The Items Schema",
        "default": "",
        "examples": [
          "SELECT * from BOOKS"
        ],
        "pattern": "^(.*)$"
      }
    },
    "variableMap": {
      "$id": "#/properties/variableMap",
      "type": "object",
      "title": "The Variablemap Schema"
    },
    "variableTypeHintMap": {
      "$id": "#/properties/variableTypeHintMap",
      "type": "object",
      "title": "The variableTypeHintMap Schema"
    }
  }
}

```

Di seguito è riportato un esempio del modello di mappatura delle richieste con una query statica:

```

{
  "version": "2018-05-29",
  "statements": [
    "select title, isbn13 from BOOKS where author = 'Mark Twain'"
  ]
}

```

Versione

Comune a tutti i modelli di mappatura delle richieste, il campo della versione definisce la versione utilizzata dal modello. Il campo della versione è obbligatorio. Il valore «2018-05-29» è l'unica versione supportata per i modelli di mappatura Amazon RDS.

```
"version": "2018-05-29"
```

Dichiarazioni e VariableMap

L'array `statements` è un segnaposto per le query fornite dallo sviluppatore. Attualmente sono supportate fino a due query per modello di mappatura delle richieste. `variableMap` è un campo facoltativo che contiene alias che possono essere utilizzati per rendere le istruzioni SQL più brevi e più leggibili. Ad esempio, è possibile quanto segue:

```
{
  "version": "2018-05-29",
  "statements": [
    "insert into BOOKS VALUES (:AUTHOR, :TITLE, :ISBN13)",
    "select * from BOOKS WHERE isbn13 = :ISBN13"
  ],
  "variableMap": {
    ":AUTHOR": $util.toJson($ctx.args.newBook.author),
    ":TITLE": $util.toJson($ctx.args.newBook.title),
    ":ISBN13": $util.toJson($ctx.args.newBook.isbn13)
  }
}
```

AWS AppSync utilizzerà i valori delle mappe variabili per creare le [SqlParameterized](#) query che verranno inviate all'API Amazon Aurora Serverless Data. Le istruzioni SQL vengono eseguite con i parametri forniti nella mappa variabile, il che elimina il rischio di iniezione SQL.

VariableTypeHintMap

`variableTypeHintMap` è un campo opzionale contenente tipi di alias che possono essere utilizzati per inviare suggerimenti sui tipi di [parametri SQL](#). Questi suggerimenti di tipo evitano l'inserimento esplicito nelle istruzioni SQL, rendendole più brevi. Ad esempio, è possibile quanto segue:

```
{
```

```
"version": "2018-05-29",
"statements": [
  "insert into LOGINDATA VALUES (:ID, :TIME)",
  "select * from LOGINDATA WHERE id = :ID"
],
"variableMap": {
  ":ID": $util.toJson($ctx.args.id),
  ":TIME": $util.toJson($ctx.args.time)
},
"variableTypeHintMap": {
  ":id": "UUID",
  ":time": "TIME"
}
}
```

AWS AppSync utilizzerà il valore della mappa variabile per creare le query inviate all'API Amazon Aurora Serverless Data. Inoltre utilizza i `variableTypeHintMap` dati e invia le informazioni del tipo a RDS. [La versione supportata da RDS typeHints è disponibile qui.](#)

AWS AppSync riferimento al modello di mappatura del resolver per OpenSearch

Note

Ora supportiamo principalmente il runtime APPSYNC_JS e la relativa documentazione. [Prendi in considerazione l'utilizzo del runtime APPSYNC_JS e delle relative guide qui.](#)

Il AWS AppSync resolver per Amazon OpenSearch Service ti consente di utilizzare GraphQL per archiviare e recuperare dati nei domini di OpenSearch servizio esistenti nel tuo account. Questo resolver consente di mappare una richiesta GraphQL in OpenSearch entrata in una richiesta di servizio, quindi mappare la risposta OpenSearch del servizio a GraphQL. Questa sezione descrive i modelli di mappatura per le operazioni di servizio supportate. OpenSearch

Richiedi un modello di mappatura

La maggior parte dei modelli di mappatura delle richieste di OpenSearch assistenza ha una struttura comune in cui cambiano solo pochi elementi. L'esempio seguente esegue una ricerca in un dominio di OpenSearch servizio, in cui i documenti sono organizzati in base a un indice chiamato `post`.

I parametri di ricerca sono definiti nella sezione `body`, con molte delle clausole di `query` comuni definite nel campo `query`. Questo esempio esegue la ricerca di documenti contenenti "Nadia" o "Bailey", o entrambe le stringhe, nel campo `author` di un documento:

```
{
  "version":"2017-02-28",
  "operation":"GET",
  "path":"/post/_search",
  "params":{
    "headers":{},
    "queryString":{},
    "body":{
      "from":0,
      "size":50,
      "query" : {
        "bool" : {
          "should" : [
            {"match" : { "author" : "Nadia" }},
            {"match" : { "author" : "Bailey" }}
          ]
        }
      }
    }
  }
}
```

Modello di mappatura delle risposte

Come con altre fonti di dati, OpenSearch Service invia una risposta AWS AppSync che deve essere convertita in GraphQL.

La maggior parte delle query GraphQL cerca il `_source` campo in una OpenSearch risposta del servizio. Poiché è possibile eseguire ricerche per restituire un singolo documento o un elenco di documenti, in Service vengono utilizzati due modelli di mappatura delle risposte comuni: OpenSearch

Elenco di risultati

```
[
  #foreach($entry in $context.result.hits.hits)
    #if( $velocityCount > 1 ) , #end
    $utils.toJson($entry.get("_source"))
  #end
```

```
]
```

Singola voce

```
$utils.toJson($context.result.get("_source"))
```

Campo **operation**

Note

Questo vale solo per il modello di mappatura Request.

Metodo o verbo HTTP (GET, POST, PUT, HEAD o DELETE) che AWS AppSync invia al dominio del OpenSearch servizio. Sia la chiave che il valore devono essere stringhe.

```
"operation" : "PUT"
```

Campo **path**

Note

Questo vale solo per il modello di mappatura Request.

Il percorso di ricerca per una richiesta OpenSearch di servizio da AWS AppSync. Costituisce un URL per il verbo HTTP dell'operazione. Sia la chiave che il valore devono essere stringhe.

```
"path" : "/<indexname>/_doc/<_id>"  
"path" : "/<indexname>/_doc"  
"path" : "/<indexname>/_search"  
"path" : "/<indexname>/_update/<_id>"
```

Quando il modello di mappatura viene valutato, questo percorso viene inviato come parte della richiesta HTTP, incluso il dominio del OpenSearch servizio. L'esempio precedente potrebbe diventare:

```
GET https://opensearch-domain-name.REGION.es.amazonaws.com/indexname/type/_search
```

Campo **params**

Note

Questo vale solo per il modello di mappatura Request.

Usato per specificare l'operazione eseguita dalla ricerca, in genere impostando il valore query all'interno di body. Ci sono tuttavia numerose altre funzionalità che è possibile configurare, ad esempio la formattazione delle risposte.

- headers

Informazioni dell'intestazione, come coppie chiave-valore. Sia la chiave che il valore devono essere stringhe. Esempio:

```
"headers" : {  
  "Content-Type" : "application/json"  
}
```

Note

AWS AppSync attualmente supporta solo JSON come Content-Type

- queryString

Coppie chiave-valore che specificano opzioni comuni, ad esempio la formattazione del codice per le risposte JSON. Sia la chiave che il valore devono essere stringhe. Ad esempio, per ottenere codice JSON con formattazione Pretty, usa:

```
"queryString" : {  
  "pretty" : "true"  
}
```

- body

Questa è la parte principale della richiesta, che consente di AWS AppSync creare una richiesta di ricerca ben formata per il dominio del OpenSearch Servizio. La chiave deve essere una stringa costituita da un oggetto. Di seguito sono illustrati un paio di esempi.

Esempio 1

Restituisce tutti i documenti in cui la città corrisponde a "seattle":

```
"body":{
  "from":0,
  "size":50,
  "query" : {
    "match" : {
      "city" : "seattle"
    }
  }
}
```

Esempio 2

Restituisce tutti i documenti in cui la città o lo stato corrisponde a "washington":

```
"body":{
  "from":0,
  "size":50,
  "query" : {
    "multi_match" : {
      "query" : "washington",
      "fields" : ["city", "state"]
    }
  }
}
```

Passare variabili

Note

Questo vale solo per il modello di mappatura Request.

È inoltre possibile passare le variabili come parte della valutazione nell'istruzione VTL. Supponi ad esempio di avere una query GraphQL come la seguente:

```
query {
  searchForState(state: "washington"){
    ...
  }
}
```



```
}  
}
```

Il modello di mappatura può accettare lo stato come argomento:

```
"body":{  
  "from":0,  
  "size":50,  
  "query" : {  
    "multi_match" : {  
      "query" : "$context.arguments.state",  
      "fields" : ["city", "state"]  
    }  
  }  
}
```

Per un elenco di utilità che è possibile includere in VTL, consulta l'argomento relativo all'[accesso alle intestazioni di richiesta](#).

AWS AppSync riferimento al modello di mappatura del resolver per Lambda

Note

Ora supportiamo principalmente il runtime APPSYNC_JS e la relativa documentazione. [Prendi in considerazione l'utilizzo del runtime APPSYNC_JS e delle relative guide qui.](#)

Puoi usare AWS AppSync funzioni e resolver per richiamare le funzioni Lambda presenti nel tuo account. Puoi modellare i payload della richiesta e la risposta delle tue funzioni Lambda prima di restituirli ai tuoi clienti. Puoi anche utilizzare modelli di mappatura per fornire suggerimenti AWS AppSync sulla natura dell'operazione da richiamare. Questa sezione descrive i diversi modelli di mappatura per le operazioni Lambda supportate.

Richiedi un modello di mappatura

Il modello di mappatura delle richieste Lambda gestisce i campi relativi alla tua funzione Lambda:

```
{
```

```
"version": string,  
"operation": Invoke|BatchInvoke,  
"payload": any type,  
"invocationType": RequestResponse|Event  
}
```

Questa è la rappresentazione dello schema JSON del modello di mappatura delle richieste Lambda una volta risolto:

```
{  
  "definitions": {},  
  "$schema": "https://json-schema.org/draft-06/schema#",  
  "$id": "https://aws.amazon.com/appsync/request-mapping-template.json",  
  "type": "object",  
  "properties": {  
    "version": {  
      "$id": "/properties/version",  
      "type": "string",  
      "enum": [  
        "2018-05-29"  
      ],  
      "title": "The Mapping template version.",  
      "default": "2018-05-29"  
    },  
    "operation": {  
      "$id": "/properties/operation",  
      "type": "string",  
      "enum": [  
        "Invoke",  
        "BatchInvoke"  
      ],  
      "title": "The Mapping template operation.",  
      "description": "What operation to execute.",  
      "default": "Invoke"  
    },  
    "payload": {},  
    "invocationType": {  
      "$id": "/properties/invocationType",  
      "type": "string",  
      "enum": [  
        "RequestResponse",  
        "Event"  
      ],  
    },  
  },  
}
```

```

    "title": "The Mapping template invocation type.",
    "description": "What invocation type to execute.",
    "default": "RequestResponse"
  }
},
"required": [
  "version",
  "operation"
],
"additionalProperties": false
}

```

Ecco un esempio che utilizza un'invokeoperazione i cui dati di payload sono il `getPost` campo di uno schema GraphQL insieme ai suoi argomenti dal contesto:

```

{
  "version": "2018-05-29",
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "arguments": $util.toJson($context.arguments)
  }
}

```

L'intero documento di mappatura viene passato come input alla funzione Lambda in modo che l'esempio precedente abbia ora il seguente aspetto:

```

{
  "version": "2018-05-29",
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "arguments": {
      "id": "postId1"
    }
  }
}

```

Versione

Comune a tutti i modelli di mappatura delle richieste, `version` definisce la versione utilizzata dal modello. `version` È obbligatorio ed è un valore statico:

```
"version": "2018-05-29"
```

Operation

L'origine dati Lambda consente di definire due operazioni sul `operation` campo: `Invoke` e `BatchInvoke`. L'`Invoke` operazione consente di AWS AppSync sapere di chiamare la funzione Lambda per ogni resolver di campi GraphQL. `BatchInvoke` indica di eseguire AWS AppSync in batch le richieste per il campo GraphQL corrente. Il campo `operation` è obbligatorio.

Infatti `Invoke`, il modello di mappatura delle richieste risolte corrisponde al payload di input della funzione Lambda. Modifichiamo l'esempio precedente:

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
  "payload": {
    "arguments": $util.toJson($context.arguments)
  }
}
```

Questo problema viene risolto e passato alla funzione Lambda, che potrebbe assomigliare a questa:

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
  "payload": {
    "arguments": {
      "id": "postId1"
    }
  }
}
```

Infatti `BatchInvoke`, il modello di mappatura viene applicato a tutti i risolutori di campo del batch. Per motivi di concisione, AWS AppSync unisce tutti i `payload` valori del modello di mappatura risolti in un elenco sotto un singolo oggetto corrispondente al modello di mappatura. Il modello di esempio seguente mostra l'unione:

```
{
  "version": "2018-05-29",
  "operation": "BatchInvoke",
  "payload": $util.toJson($context)
```

```
}
```

Questo modello viene risolto nel documento di mappatura seguente:

```
{
  "version": "2018-05-29",
  "operation": "BatchInvoke",
  "payload": [
    {...}, // context for batch item 1
    {...}, // context for batch item 2
    {...}  // context for batch item 3
  ]
}
```

Ogni elemento dell'`payload` elenco corrisponde a un singolo elemento del batch. La funzione Lambda dovrebbe inoltre restituire una risposta a forma di elenco corrispondente all'ordine degli elementi inviati nella richiesta:

```
[
  { "data": {...}, "errorMessage": null, "errorType": null }, // result for batch item 1
  { "data": {...}, "errorMessage": null, "errorType": null }, // result for batch item 2
  { "data": {...}, "errorMessage": null, "errorType": null }  // result for batch item 3
]
```

Carico utile

Il `payload` campo è un contenitore utilizzato per passare qualsiasi JSON ben formato alla funzione Lambda. Se il `operation` campo è impostato su `BatchInvoke`, AWS AppSync racchiude i valori esistenti `payload` in un elenco. Il campo `payload` è facoltativo.

Tipo di invocazione

L'origine dati Lambda consente di definire due tipi di invocazione: e. `RequestResponse Event` [I tipi di invocazione sono sinonimi dei tipi di invocazione definiti nell'API Lambda](#). Il tipo di `RequestResponse` invocazione consente di AWS AppSync chiamare la funzione Lambda in modo sincrono per attendere una risposta. L'`Event` invocazione consente di richiamare la funzione Lambda in modo asincrono. [Per ulteriori informazioni su come Lambda gestisce le richieste dei tipi di Event invocazione, consulta Invocazione asincrona](#). Il campo `invocationType` è facoltativo. Se questo

campo non è incluso nella richiesta, verrà utilizzato per impostazione predefinita il tipo di invocazione. **AWS AppSync RequestResponse**

Per qualsiasi `invocationType` campo, la richiesta risolta corrisponde al payload di input della funzione Lambda. Modifichiamo l'esempio precedente:

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
  "invocationType": "Event"
  "payload": {
    "arguments": $util.toJson($context.arguments)
  }
}
```

Questo problema viene risolto e passato alla funzione Lambda, che potrebbe assomigliare a questa:

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
  "invocationType": "Event",
  "payload": {
    "arguments": {
      "id": "postId1"
    }
  }
}
```

Quando l'`BatchInvoke` operazione viene utilizzata insieme al campo del tipo di `Event` invocazione, AWS AppSync unisce il resolver di campo nello stesso modo menzionato sopra e la richiesta viene passata alla funzione Lambda come evento asincrono costituito da un elenco di valori. `payload` Ti consigliamo di disabilitare la memorizzazione nella cache dei resolver per i resolver di tipo di `Event` invocazione perché questi non verrebbero inviati a Lambda se si verificasse un problema nella cache.

Modello di mappatura delle risposte

Come con altre fonti di dati, la funzione Lambda invia una risposta a AWS AppSync che deve essere convertita in un tipo GraphQL.

Il risultato della funzione Lambda viene impostato sull'`context` oggetto disponibile tramite la proprietà `Velocity Template Language (VTL)`. `$context.result`

Se la forma della risposta della funzione Lambda corrisponde esattamente alla forma del tipo GraphQL, puoi inoltrare la risposta usando il modello di mappatura della risposta seguente:

```
$util.toJson($context.result)
```

Non ci sono campi obbligatori né restrizioni relative alla forma che si applicano al modello di mappatura della risposta. Tuttavia, poiché GraphQL è un protocollo fortemente tipizzato, il modello di mappatura risolto deve corrispondere al tipo GraphQL previsto.

Risposta in batch della funzione Lambda

Se il campo `operation` è impostato su `BatchInvoke`, AWS AppSync si aspetta che la funzione Lambda restituisca un elenco di voci. Per riportare ogni risultato AWS AppSync all'elemento della richiesta originale, l'elenco delle risposte deve corrispondere per dimensioni e ordine. È valido avere `null` elementi nell'elenco delle risposte; `$ctx.result` è impostato di conseguenza su `null`.

Resolver Lambda diretti

Se desideri aggirare completamente l'uso dei modelli di mappatura, AWS AppSync puoi fornire un payload predefinito alla tua funzione Lambda e una risposta della funzione Lambda predefinita a un tipo GraphQL. Puoi scegliere di fornire un modello di richiesta, un modello di risposta o nessuno dei due e gestirlo di conseguenza. AWS AppSync

Modello di mappatura delle richieste Direct Lambda

Quando il modello di mappatura della richiesta non viene fornito, AWS AppSync invierà l'oggetto `Context` direttamente alla funzione Lambda come `Invoke` operazione. Per ulteriori informazioni sulla struttura dell'oggetto `Context`, consulta [AWS AppSync riferimento al contesto del modello di mappatura del resolver](#).

Modello di mappatura della risposta Direct Lambda

Quando il modello di mappatura delle risposte non viene fornito, AWS AppSync esegue una delle due operazioni dopo aver ricevuto la risposta della funzione Lambda. Se non hai fornito un modello di mappatura delle richieste o se hai fornito un modello di mappatura delle richieste con la versione 2018-05-29, la risposta sarà equivalente al seguente modello di mappatura delle risposte:

```
#if($ctx.error)
    $util.error($ctx.error.message, $ctx.error.type, $ctx.result)
```

```
#end
$util.toJson($ctx.result)
```

Se hai fornito un modello con la versione 2017-02-28, la logica di risposta funziona in modo equivalente al seguente modello di mappatura delle risposte:

```
$util.toJson($ctx.result)
```

Superficialmente, il bypass del modello di mappatura funziona in modo simile all'utilizzo di determinati modelli di mappatura, come mostrato negli esempi precedenti. Tuttavia, dietro le quinte, la valutazione dei modelli di mappatura viene completamente elusa. Poiché la fase di valutazione del modello viene ignorata, in alcuni scenari le applicazioni potrebbero subire un sovraccarico e una latenza inferiori durante la risposta rispetto a una funzione Lambda con un modello di mappatura delle risposte che deve essere valutato.

Gestione personalizzata degli errori nelle risposte di Direct Lambda Resolver

Puoi personalizzare le risposte agli errori delle funzioni Lambda richiamate dai Direct Lambda Resolver sollevando un'eccezione personalizzata. L'esempio seguente mostra come creare un'eccezione personalizzata utilizzando: JavaScript

```
class CustomException extends Error {
  constructor(message) {
    super(message);
    this.name = "CustomException";
  }
}

throw new CustomException("Custom message");
```

Quando vengono sollevate eccezioni, i `errorType` e `errorMessage` sono rispettivamente l'errore personalizzato generato `name` e `message` l'errore personalizzato generato.

In caso `errorType UnauthorizedException` AWS AppSync affermativo, restituisce il messaggio predefinito ("You are not authorized to make this call.") anziché un messaggio personalizzato.

Il seguente frammento è un esempio di risposta GraphQL che dimostra una personalizzazione: `errorType`


```
{
  "data": {
    "query": null
  },
  "errors": [
    {
      "path": [
        "query"
      ],
      "data": null,
      "errorType": "CustomException",
      "errorInfo": null,
      "locations": [
        {
          "line": 5,
          "column": 10,
          "sourceName": null
        }
      ],
      "message": "Custom Message"
    }
  ]
}
```

Direct Lambda Resolver: batch abilitato

Puoi abilitare il batching per il tuo Direct Lambda Resolver `maxBatchSize` configurandolo sul tuo resolver. Quando `maxBatchSize` è impostato su un valore maggiore di quello `0` di un resolver Direct Lambda, AWS AppSync invia richieste in batch alla funzione Lambda con dimensioni fino a `maxBatchSize`.

L'impostazione `maxBatchSize 0` su un resolver Direct Lambda disattiva il batch.

Per ulteriori informazioni su come funziona il batching con i resolver Lambda, consulta. [Caso d'uso avanzato: Batching](#)

Richiedi un modello di mappatura

Quando il batching è abilitato e il modello di mappatura della richiesta non viene fornito, AWS AppSync invia un elenco di Context oggetti come `BatchInvoke` operazione direttamente alla funzione Lambda.

Modello di mappatura delle risposte

Quando il batching è abilitato e il modello di mappatura delle risposte non viene fornito, la logica di risposta è equivalente al seguente modello di mappatura delle risposte:

```
#if( $context.result && $context.result.errorMessage )
    $utils.error($context.result.errorMessage, $context.result.errorType,
    $context.result.data)
#else
    $utils.toJson($context.result.data)
#end
```

La funzione Lambda deve restituire un elenco di risultati nello stesso ordine dell'elenco degli Context oggetti inviati. È possibile restituire singoli errori fornendo un `errorMessage` e `errorType` per un risultato specifico. Ogni risultato dell'elenco ha il seguente formato:

```
{
  "data" : { ... }, // your data
  "errorMessage" : { ... }, // optional, if included an error entry is added to the
  "errors" object in the AppSync response
  "errorType" : { ... } // optional, the error type
}
```

Note

Gli altri campi dell'oggetto risultato sono attualmente ignorati.

Gestione degli errori da Lambda

Puoi restituire un errore per tutti i risultati generando un'eccezione o un errore nella tua funzione Lambda. Se la dimensione della richiesta di payload o della risposta per la richiesta batch è troppo grande, Lambda restituisce un errore. In tal caso, dovresti prendere in considerazione la possibilità di ridurre `maxBatchSize` o ridurre le dimensioni del payload di risposta.

Per informazioni sulla gestione dei singoli errori, vedere [Restituzione di errori individuali](#).

Funzioni Lambda di esempio

Utilizzando lo schema seguente, puoi creare un Direct Lambda Resolver per il `Post.relatedPosts` field resolver e abilitare il batching impostando sopra: `maxBatchSize 0`

```
schema {
  query: Query
  mutation: Mutation
}

type Query {
  getPost(id:ID!): Post
  allPosts: [Post]
}

type Mutation {
  addPost(id: ID!, author: String!, title: String, content: String, url: String):
  Post!
}

type Post {
  id: ID!
  author: String!
  title: String
  content: String
  url: String
  ups: Int
  downs: Int
  relatedPosts: [Post]
}
```

Nella seguente query, la funzione Lambda verrà chiamata con batch di richieste da risolvere: `relatedPosts`

```
query getAllPosts {
  allPosts {
    id
    relatedPosts {
      id
    }
  }
}
```

Di seguito viene fornita una semplice implementazione di una funzione Lambda:

```
const posts = {
  1: {
```

```

    id: '1',
    title: 'First book',
    author: 'Author1',
    url: 'https://amazon.com/',
    content:
      'SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT
AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1',
    ups: '100',
    downs: '10',
  },
  2: {
    id: '2',
    title: 'Second book',
    author: 'Author2',
    url: 'https://amazon.com',
    content: 'SAMPLE TEXT AUTHOR 2 SAMPLE TEXT AUTHOR 2 SAMPLE TEXT',
    ups: '100',
    downs: '10',
  },
  3: { id: '3', title: 'Third book', author: 'Author3', url: null, content: null, ups:
null, downs: null },
  4: {
    id: '4',
    title: 'Fourth book',
    author: 'Author4',
    url: 'https://www.amazon.com/',
    content:
      'SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
AUTHOR 4',
    ups: '1000',
    downs: '0',
  },
  5: {
    id: '5',
    title: 'Fifth book',
    author: 'Author5',
    url: 'https://www.amazon.com/',
    content: 'SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE
TEXT AUTHOR 5 SAMPLE TEXT',
    ups: '50',
    downs: '0',
  },
}

```

```
const relatedPosts = {
  1: [posts['4']],
  2: [posts['3'], posts['5']],
  3: [posts['2'], posts['1']],
  4: [posts['2'], posts['1']],
  5: [],
}

exports.handler = async (event) => {
  console.log('event ->', event)
  // retrieve the ID of each post
  const ids = event.map((context) => context.source.id)
  // fetch the related posts for each post id
  const related = ids.map((id) => relatedPosts[id])

  // return the related posts; or an error if none were found
  return related.map((r) => {
    if (r.length > 0) {
      return { data: r }
    } else {
      return { data: null, errorMessage: 'Not found', errorType: 'ERROR' }
    }
  })
}
```

AWS AppSync riferimento al modello di mappatura del resolver per EventBridge

Note

Ora supportiamo principalmente il runtime APPSYNC_JS e la relativa documentazione.
[Prendi in considerazione l'utilizzo del runtime APPSYNC_JS e delle relative guide qui.](#)

Il modello di mappatura del AWS AppSync resolver utilizzato con l'origine EventBridge dati consente di inviare eventi personalizzati al bus Amazon. EventBridge

Richiedi un modello di mappatura

Il modello di mappatura delle PutEvents richieste consente di inviare più eventi personalizzati a un EventBridge bus di eventi. Il documento di mappatura ha la seguente struttura:

```
{
  "version" : "2018-05-29",
  "operation" : "PutEvents",
  "events" : [{}]
```

Di seguito è riportato un esempio di modello di mappatura delle richieste per: EventBridge

```
{
  "version": "2018-05-29",
  "operation": "PutEvents",
  "events": [{
    "source": "com.mycompany.myapp",
    "detail": {
      "key1" : "value1",
      "key2" : "value2"
    },
    "detailType": "myDetailType1"
  },
  {
    "source": "com.mycompany.myapp",
    "detail": {
      "key3" : "value3",
      "key4" : "value4"
    },
    "detailType": "myDetailType2",
    "resources" : ["Resource1", "Resource2"],
    "time" : "2023-01-01T00:30:00.000Z"
  }
]
```

Modello di mappatura delle risposte

Se l'PutEventsoperazione ha esito positivo, la risposta di EventBridge è inclusa in: `$ctx.result`

```

#if($ctx.error)
  $util.error($ctx.error.message, $ctx.error.type, $ctx.result)
#end
$util.toJson($ctx.result)

```

Errori che si verificano durante l'esecuzione di PutEvents operazioni come InternalExceptions o Timeouts che verranno visualizzati in `$ctx.error`. Per un elenco degli EventBridge errori più comuni, consulta il [riferimento agli errori EventBridge comuni](#).

`result` Sarà nel seguente formato:

```

{
  "Entries" [
    {
      "ErrorCode" : String,
      "ErrorMessage" : String,
      "EventId" : String
    }
  ],
  "FailedEntryCount" : number
}

```

- **Iscrizioni**

I risultati dell'evento ingerito, sia riusciti che infruttuosi. Se l'ingestione è andata a buon fine, la voce contiene il EventID. In caso contrario, è possibile utilizzare il comando ErrorCode and ErrorMessage per identificare il problema relativo alla voce.

Per ogni record, l'indice dell'elemento di risposta è lo stesso dell'indice nell'array di richiesta.

- **FailedEntryCount**

Il numero di inserimenti non riusciti. Questo valore è rappresentato come un numero intero.

Per ulteriori informazioni sulla risposta di PutEvents, vedere [PutEvents](#).

Esempio di risposta 1

L'esempio seguente è un PutEvent operazione con due eventi riusciti:

```

{

```

```
"Entries" : [  
  {  
    "EventId": "11710aed-b79e-4468-a20b-bb3c0c3b4860"  
  },  
  {  
    "EventId": "d804d26a-88db-4b66-9eaf-9a11c708ae82"  
  }  
],  
"FailedEntryCount" : 0  
}
```

Esempio di risposta di esempio 2

L'esempio seguente è un'PutEvent operazione con tre eventi, due riusciti e uno fallito:

```
{  
  "Entries" : [  
    {  
      "EventId": "11710aed-b79e-4468-a20b-bb3c0c3b4860"  
    },  
    {  
      "EventId": "d804d26a-88db-4b66-9eaf-9a11c708ae82"  
    },  
    {  
      "ErrorCode" : "SampleErrorCode",  
      "ErrorMessage" : "Sample Error Message"  
    }  
  ],  
  "FailedEntryCount" : 1  
}
```

PutEvents campi

PutEvents contiene i seguenti campi del modello di mappatura:

- Versione

Comune a tutti i modelli di mappatura delle richieste, il `version` campo definisce la versione utilizzata dal modello. Questo campo è obbligatorio. Il valore `2018-05-29` è l'unica versione supportata per i modelli di EventBridge mappatura.

- Operazioni

L'unica operazione supportata è `PutEvents`. Questa operazione consente di aggiungere eventi personalizzati al bus degli eventi.

- Eventi

Una serie di eventi che verranno aggiunti all'event bus. Questo array dovrebbe avere un'allocazione di 1-10 elementi.

L'Evento oggetto è un oggetto JSON valido con i seguenti campi:

- `"source"`: Una stringa che definisce l'origine dell'evento.
- `"detail"`: un oggetto JSON che è possibile utilizzare per allegare informazioni sull'evento. Questo campo può essere una mappa vuota (`{ }`).
- `"detailType"`: Una stringa che identifica il tipo di evento.
- `"resources"`: Un array di stringhe JSON che identifica le risorse coinvolte nell'evento. Questo campo può essere una matrice vuota.
- `"time"`: Il timestamp dell'evento fornito come stringa. Questo dovrebbe seguire il formato del [RFC3339](#) timestamp.

I frammenti seguenti sono alcuni esempi di oggetti validi: Event

Esempio 1

```
{
  "source" : "source1",
  "detail" : {
    "key1" : [1,2,3,4],
    "key2" : "strval"
  },
  "detailType" : "sampleDetailType",
  "resources" : ["Resource1", "Resource2"],
  "time" : "2022-01-10T05:00:10Z"
}
```

Esempio 2

```
{
  "source" : "source1",
  "detail" : {},
  "detailType" : "sampleDetailType"
}
```

```
}
```

Esempio 3

```
{
  "source" : "source1",
  "detail" : {
    "key1" : 1200
  },
  "detailType" : "sampleDetailType",
  "resources" : []
}
```

AWS AppSync riferimento al modello di mappatura del resolver per l'origine dati **None**

Note

Ora supportiamo principalmente il runtime APPSYNC_JS e la relativa documentazione. [Prendi in considerazione l'utilizzo del runtime APPSYNC_JS e delle relative guide qui.](#)

Il modello di mappatura del AWS AppSync resolver utilizzato con l'origine dati di tipo None consente di modellare le richieste per le operazioni locali. AWS AppSync

Richiedi un modello di mappatura

Il modello di mappatura è semplice e permette di passare quante più informazioni sul contesto possibile tramite il campo `payload`.

```
{
  "version": string,
  "payload": any type
}
```

Di seguito è riportata la rappresentazione dello schema JSON del modello di mappatura della richiesta, dopo la risoluzione:

```
{
  "definitions": {},
  "$schema": "https://json-schema.org/draft-06/schema#",
  "$id": "https://aws.amazon.com/appsync/request-mapping-template.json",
  "type": "object",
  "properties": {
    "version": {
      "$id": "/properties/version",
      "type": "string",
      "enum": [
        "2018-05-29"
      ],
      "title": "The Mapping template version.",
      "default": "2018-05-29"
    },
    "payload": {}
  },
  "required": [
    "version"
  ],
  "additionalProperties": false
}
```

Ecco un esempio in cui gli argomenti del campo vengono passati tramite la proprietà di contesto VTL: `$context.arguments`

```
{
  "version": "2018-05-29",
  "payload": $util.toJson($context.arguments)
}
```

Il valore del campo `payload` verrà inoltrato al modello di mappatura della risposta e sarà disponibile nella proprietà di contesto VTL (`$context.result`).

Questo è un esempio che rappresenta il valore interpolato del campo `payload`:

```
{
  "id": "postId1"
}
```

Versione

Comune a tutti i modelli di mappatura delle richieste, il `version` campo definisce la versione utilizzata dal modello.

Il campo `version` è obbligatorio.

Esempio:

```
"version": "2018-05-29"
```

Carico utile

Il campo `payload` è un container che può essere usato per passare qualsiasi input JSON con formato corretto al modello di mappatura della risposta.

Il campo `payload` è facoltativo.

Modello di mappatura delle risposte

Poiché non è presente alcuna origine dati, il valore del campo `payload` verrà inoltrato al modello di mappatura della risposta e impostato sull'oggetto `context` che è disponibile tramite la proprietà `$context.result` VTL.

Se la forma del valore del campo `payload` corrisponde esattamente alla forma del tipo GraphQL, puoi inoltrare la risposta usando il modello di mappatura della risposta seguente:

```
$util.toJson($context.result)
```

Non ci sono campi obbligatori né restrizioni relative alla forma che si applicano al modello di mappatura della risposta. Tuttavia, poiché GraphQL è un protocollo fortemente tipizzato, il modello di mappatura risolto deve corrispondere al tipo GraphQL previsto.

AWS AppSync riferimento al modello di mappatura del resolver per HTTP

Note

Ora supportiamo principalmente il runtime APPSYNC_JS e la relativa documentazione.

[Prendi in considerazione l'utilizzo del runtime APPSYNC_JS e delle relative guide qui.](#)

I modelli di mappatura del resolver AWS AppSync HTTP consentono di inviare richieste da qualsiasi endpoint HTTP e AWS AppSync di inviare risposte dall'endpoint HTTP a. AWS AppSync Utilizzando i modelli di mappatura, è possibile fornire suggerimenti AWS AppSync sulla natura dell'operazione da richiamare. In questa sezione vengono descritti i diversi modelli di mappatura per i resolver HTTP supportati.

Richiedi un modello di mappatura

```
{
  "version": "2018-05-29",
  "method": "PUT|POST|GET|DELETE|PATCH",
  "params": {
    "query": Map,
    "headers": Map,
    "body": any
  },
  "resourcePath": string
}
```

Dopo che il modello di mappatura della richiesta HTTP è stato risolto, la rappresentazione dello schema JSON del modello di mappatura della richiesta è simile alla seguente:

```
{
  "$id": "https://aws.amazon.com/appsync/request-mapping-template.json",
  "type": "object",
  "properties": {
    "version": {
      "$id": "/properties/version",
      "type": "string",
      "title": "The Version Schema ",
      "default": "",

```

```

    "examples": [
      "2018-05-29"
    ],
    "enum": [
      "2018-05-29"
    ]
  },
  "method": {
    "$id": "/properties/method",
    "type": "string",
    "title": "The Method Schema ",
    "default": "",
    "examples": [
      "PUT|POST|GET|DELETE|PATCH"
    ],
    "enum": [
      "PUT",
      "PATCH",
      "POST",
      "DELETE",
      "GET"
    ]
  },
  "params": {
    "$id": "/properties/params",
    "type": "object",
    "properties": {
      "query": {
        "$id": "/properties/params/properties/query",
        "type": "object"
      },
      "headers": {
        "$id": "/properties/params/properties/headers",
        "type": "object"
      },
      "body": {
        "$id": "/properties/params/properties/body",
        "type": "string",
        "title": "The Body Schema ",
        "default": "",
        "examples": [
          ""
        ]
      }
    }
  }
}

```

```

    }
  },
  "resourcePath": {
    "$id": "/properties/resourcePath",
    "type": "string",
    "title": "The Resourcepath Schema ",
    "default": "",
    "examples": [
      ""
    ]
  }
},
"required": [
  "version",
  "method",
  "resourcePath"
]
}

```

Di seguito è riportato un esempio di richiesta HTTP POST con un corpo text/plain:

```

{
  "version": "2018-05-29",
  "method": "POST",
  "params": {
    "headers": {
      "Content-Type": "text/plain"
    },
    "body": "this is an example of text body"
  },
  "resourcePath": "/"
}

```

Versione

Note

Questo vale solo per il modello di mappatura Request.

Definisce la versione usata dal modello. `version` è comune a tutti i modelli di mappatura della richiesta ed è obbligatorio.

```
"version": "2018-05-29"
```

Metodo

Note

Questo vale solo per il modello di mappatura Request.

Metodo o verbo HTTP (GET, POST, PUT, PATCH o DELETE) che AWS AppSync invia all'endpoint HTTP.

```
"method": "PUT"
```

ResourcePath

Note

Questo vale solo per il modello di mappatura Request.

Il percorso delle risorse a cui si desidera accedere. Oltre all'endpoint nell'origine dati HTTP, il percorso della risorsa costituisce l'URL a cui il AWS AppSync servizio effettua una richiesta.

```
"resourcePath": "/v1/users"
```

Quando il modello di mappatura viene valutato, questo percorso viene inviato come parte della richiesta HTTP, incluso l'endpoint HTTP. Ad esempio, l'esempio precedente potrebbe diventare il seguente:

```
PUT <endpoint>/v1/users
```

Campi dei parametri

Note

Questo vale solo per il modello di mappatura Request.

Usato per specificare l'operazione eseguita dalla ricerca, in genere impostando il valore query all'interno di body. Ci sono tuttavia numerose altre funzionalità che è possibile configurare, ad esempio la formattazione delle risposte.

headers

Informazioni dell'intestazione, come coppie chiave-valore. Sia la chiave che il valore devono essere stringhe.

Esempio:

```
"headers" : {  
  "Content-Type" : "application/json"  
}
```

Le intestazioni Content-Type attualmente supportate sono:

```
text/*  
application/xml  
application/json  
application/soap+xml  
application/x-amz-json-1.0  
application/x-amz-json-1.1  
application/vnd.api+json  
application/x-ndjson
```

Nota: non è possibile impostare le seguenti intestazioni HTTP:

```
HOST  
CONNECTION  
USER-AGENT  
EXPECTATION  
TRANSFER_ENCODING  
CONTENT_LENGTH
```

query

Coppie chiave-valore che specificano opzioni comuni, ad esempio la formattazione del codice per le risposte JSON. Sia la chiave che il valore devono essere stringhe. L'esempio seguente mostra in che modo è possibile inviare una stringa di query come ?type=json:

```
"query" : {
```

```
{
  "type" : "json"
}
```

body

Il corpo contiene il corpo della richiesta HTTP che si decide di impostare. La richiesta corpo è sempre una stringa con codifica UTF-8, a meno che il tipo di contenuto non specifichi il charset.

```
"body": "body string"
```

Risposta

Consulta un esempio [qui](#).

Autorità di certificazione (CA) riconosciute dagli AWS AppSync endpoint HTTPS

Note

Let's Encrypt è accettato tramite i certificati identrust e isrgrootx1. Non è richiesta alcuna azione da parte tua se usi Let's Encrypt.

Al momento, i certificati autofirmati non sono supportati dai resolver HTTP quando si utilizza HTTPS. AWS AppSync riconosce le seguenti autorità di certificazione durante la risoluzione dei certificati SSL/TLS per HTTPS:

Certificati root noti in AWS AppSync

Nome	Data	SHA1 Impronta digitale
digicertassuredigrootca	21 aprile 2018	05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E:4B:DF:B5:A8:99:B2:4D:43
trustcenterclass2caii	21 aprile 2018	AE:50:83:ED:7C:F4:5C:BC:8F:61:C6:21:FE:68:5D:79:42:21:15:6E
thawtepremiumserverca	21 aprile 2018	E0:AB:05:94:20:72:54:93:05:60:62:02:36:70:F7:CD:2E:FC:66:66

Nome	Data	SHA1 Impronta digitale
<code>cia-crt-g3-02-ca</code>	23 novembre 2016	96:4A:BB:A7:BD:DA:FC:97:34:C0:0A:2D:F0:05:98:F7:E6:C6:6F:09
<code>swisssignplatinumg2ca</code>	21 aprile 2018	56:E0:FA:C0:3B:8F:18:23:55:18:E5:D3:11:CA:E8:C2:43:31:AB:66
<code>swisssignsilverg2ca</code>	21 aprile 2018	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
<code>thawteserverca</code>	21 aprile 2018	9F:AD:91:A6:CE:6A:C6:C5:00:47:C4:4E:C9:D4:A5:0D:92:D8:49:79
<code>equifaxsecurebusinessca1</code>	21 aprile 2018	AE:E6:3D:70:E3:76:FB:C7:3A:EB:B0:A1:C1:D4:C4:7A:A7:40:B3:F4
<code>securetrustca</code>	21 aprile 2018	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:FF:11
<code>utnuserfirstclientauthemailca</code>	21 aprile 2018	B1:72:B1:A5:6D:95:F9:1F:E5:02:87:E1:4D:37:EA:6A:44:63:76:8A
<code>thawtepersonalfreeemailca</code>	21 aprile 2018	E6:18:83:AE:84:CA:C1:C1:CD:52:AD:E8:E9:25:2B:45:A6:4F:B7:E2
<code>affirmtrustnetworkingca</code>	21 aprile 2018	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
<code>entrustevca</code>	21 aprile 2018	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
<code>utnuserfirsthardwareca</code>	21 aprile 2018	04:83:ED:33:99:AC:36:08:05:87:22:ED:BC:5E:46:00:E3:BE:F9:D7
<code>certumca</code>	21 aprile 2018	62:52:DC:40:F7:11:43:A2:2F:DE:9E:F7:34:8E:06:42:51:B1:81:18

Nome	Data	SHA1 Impronta digitale
addtrustclass1ca	21 aprile 2018	CC:AB:0E:A0:4C:23:01:D6:69:7B:DD:37:9F:CD:12:EB:24:E3:94:9D
entrustrootcag2	21 aprile 2018	8C:F4:27:FD:79:0C:3A:D1:66:06:8D:E8:1E:57:EF:BB:93:22:72:D4
equifaxsecureca	21 aprile 2018	D2:32:09:AD:23:D3:14:23:21:74:E4:0D:7F:9D:62:13:97:86:63:3A
quovadisrootca3	21 aprile 2018	1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:51:85
quovadisrootca2	21 aprile 2018	CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7C:F7
digicertglobalrootg2	21 aprile 2018	DF:3C:24:F9:BF:D6:66:76:1B:26:80:73:FE:06:D1:CC:8D:4F:82:A4
digicerthighassuranceevrootca	21 aprile 2018	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:DC:25
secomvalicertclass1ca	21 aprile 2018	E5:DF:74:3C:B6:01:C4:9B:98:43:DC:AB:8C:E8:6A:81:10:9F:E4:8E
equifaxsecureglobalbusinessca1	21 aprile 2018	3A:74:CB:7A:47:DB:70:DE:89:1F:24:35:98:64:B8:2D:82:BD:1A:36
geotrustuniversalca	21 aprile 2018	E6:21:F3:35:43:79:05:9A:4B:68:30:9D:8A:2F:74:22:15:87:EC:79
deprecateditsecca	27 gennaio 2012	12:12:0B:03:0E:15:14:54:F4:DD:B3:F5:DE:13:6E:83:5A:29:72:9D
verisignclass3ca	21 aprile 2018	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
thawteprimaryrootcag3	21 aprile 2018	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6B:F2

Nome	Data	SHA1 Impronta digitale
thawteprimaryrootcag2	21 aprile 2018	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:F0:12
deutschetelekomrootca2	21 aprile 2018	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD:D6:13:30:FD:8C:DE:37:BF
buypassclass3ca	21 aprile 2018	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
utnuserfirstobjectca	21 aprile 2018	E1:2D:FB:4B:41:D7:D9:C3:2B:30:51:4B:AC:1D:81:D8:38:5E:2D:46
geotrustprimaryca	21 aprile 2018	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
buypassclass2ca	21 aprile 2018	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99
baltimorecodesigningca	21 aprile 2018	30:46:D8:C8:88:FF:69:30:C3:4A:FC:CD:49:27:08:7C:60:56:7B:0D
verisignclass1ca	21 aprile 2018	CE:6A:64:A3:09:E4:2F:BB:D9:85:1C:45:3E:64:09:EA:E8:7D:60:F1
baltimorecybertrustca	21 aprile 2018	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
starfieldclass2ca	21 aprile 2018	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:B5:8A
camerfirmachamberscommerceca	21 aprile 2018	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1
ttelesecglobalrootclass3ca	21 aprile 2018	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
verisignclass3g5ca	21 aprile 2018	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5

Nome	Data	SHA1 Impronta digitale
ttelesecglobalrootclass2ca	21 aprile 2018	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
trustcenteruniversalc	21 aprile 2018	6B:2F:34:AD:89:58:BE:62:FD:B0:6B:5C:CE:BB:9D:D9:4F:4E:39:F3
verisignclass3g4ca	21 aprile 2018	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
verisignclass3g3ca	21 aprile 2018	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
xrampglobalca	21 aprile 2018	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
amzninternalrootca	12 dicembre 2008	A7:B7:F6:15:8A:FF:1E:C8:85:13:38:BC:93:EB:A2:AB:A4:09:EF:06
certplusclass3pprimaryca	21 aprile 2018	21:6B:2A:29:E6:2A:00:CE:82:01:46:D8:24:41:41:B9:25:11:B2:79
certumtrustednetworkca	21 aprile 2018	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28:A2:59:3A:19:A7:0F:06:9E
verisignclass3g2ca	21 aprile 2018	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F
globalsignr3ca	21 aprile 2018	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
utndatacorpsgccca	21 aprile 2018	58:11:9F:0E:12:82:87:EA:50:FD:D9:87:45:6F:4F:78:DC:FA:D6:D4
secomscrootca2	21 aprile 2018	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74

Nome	Data	SHA1 Impronta digitale
gtecybertrustglobalca	21 aprile 2018	97:81:79:50:D8:1C:96:70:CC:34:D8:09:CF:79:44:31:36:7E:F4:74
secomscrootca1	21 aprile 2018	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
affirmtrustcommercialca	21 aprile 2018	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
trustcenterclass4caii	21 aprile 2018	A6:9A:91:FD:05:7F:13:6A:42:63:0B:B1:76:0D:2D:51:12:0C:16:50
verisignuniversalrootca	21 aprile 2018	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
globalsignr2ca	21 aprile 2018	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
certplusclass2primaryca	21 aprile 2018	74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:E2:BB
digicertglobalrootca	21 aprile 2018	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:54:36
globalsignca	21 aprile 2018	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
thawteprimaryrootca	21 aprile 2018	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7B:81
starfieldrootg2ca	21 aprile 2018	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
geotrustglobalca	21 aprile 2018	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:82:12
soneraclass2ca	21 aprile 2018	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27

Nome	Data	SHA1 Impronta digitale
verisigntsaca	21 aprile 2018	20:CE:B1:F0:F5:1C:0E:19:A9:F3:8D:B1:AA:8E:03:8C:AA:7A:C7:01
soneraclass1ca	21 aprile 2018	07:47:22:01:99:CE:74:B9:7C:B0:3D:79:B2:64:A2:C8:55:E9:33:FF
quovadisrootca	21 aprile 2018	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9
affirmtrustpremium eccca	21 aprile 2018	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
starfieldservicesr ootg2ca	21 aprile 2018	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
valicertclass2ca	21 aprile 2018	31:7A:2A:D0:7F:2B:33:5E:F5:A1:C3:4E:4B:57:E8:B7:D8:F1:FC:A6
comodoaaaca	21 aprile 2018	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
aolrootca2	21 aprile 2018	85:B5:FF:67:9B:0C:79:96:1F:C8:6E:44:22:00:46:13:DB:17:92:84
keynectisrootca	21 aprile 2018	9C:61:5C:4D:4D:85:10:3A:53:26:C2:4D:BA:EA:E4:A2:D2:D5:CC:97
addtrustqualifiedc a	21 aprile 2018	4D:23:78:EC:91:95:39:B5:00:7F:75:8F:03:3B:21:1E:C5:4D:8B:CF
aolrootca1	21 aprile 2018	39:21:C1:15:C1:5D:0E:CA:5C:CB:5B:C4:F0:7D:21:D8:05:0B:56:6A
verisignclass2g3ca	21 aprile 2018	61:EF:43:D7:7F:CA:D4:61:51:BC:98:E0:C3:59:12:AF:9F:EB:63:11
addtrustexternalca	21 aprile 2018	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68

Nome	Data	SHA1 Impronta digitale
verisignclass2g2ca	21 aprile 2018	B3:EA:C4:47:76:C9:C8:1C:EA:F2:9D:95: B6:CC:A0:08:1B:67:EC:9D
geotrustprimarycag 3	21 aprile 2018	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B: 20:D2:D9:32:3A:4C:2A:FD
geotrustprimarycag 2	21 aprile 2018	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57: 8B:51:9A:99:E6:10:D7:B0
swisssigngoldg2ca	21 aprile 2018	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6: 45:25:3A:6F:9F:1A:27:61
entrust2048ca	21 aprile 2018	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB: E7:92:7D:7D:65:2D:34:31
chunghwaepkirootca	21 aprile 2018	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4: 56:4B:CF:E2:3D:69:C6:F0
camerfirmachambers ignca	21 aprile 2018	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52: A1:2C:5B:29:F6:D6:AA:0C
camerfirmachambers ca	21 aprile 2018	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50: BA:9E:A8:7E:FE:9A:CE:3C
godaddyclass2ca	21 aprile 2018	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0: D7:77:70:02:8F:20:EE:E4
affirmtrustpremium ca	21 aprile 2018	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F: 7D:6A:06:65:26:32:28:27
verisignclass1g3ca	21 aprile 2018	20:42:85:DC:F7:EB:76:41:95:57:8E:13: 6B:D4:B7:D1:E9:8E:46:A5
secomevrootca1	21 aprile 2018	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8: 90:8F:FD:28:86:65:64:7D
verisignclass1g2ca	21 aprile 2018	27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B: 56:16:7F:62:F5:32:E5:47

Nome	Data	SHA1 Impronta digitale
amzninternalinfoseccag3	27 febbraio 2015	B9:B1:CA:38:F7:BF:9C:D2:D4:95:E7:B6:5E:75:32:9B:A8:78:2E:F6
cia-crt-g3-01-ca	23 novembre 2016	2B:EE:2C:BA:A3:1D:B5:FE:60:40:41:95:08:ED:46:82:39:4D:ED:E2
godaddyrootg2ca	21 aprile 2018	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
digicertassuredidrootca	21 aprile 2018	05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E:4B:DF:B5:A8:99:B2:4D:43
microseceszignorootca2009	21 aprile 2018	89:DF:74:FE:5C:F4:0F:4A:80:F9:E3:37:7D:54:DA:91:E1:01:31:8E
affirmtrustcommercial	21 aprile 2018	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
comodoecccertificationauthority	21 aprile 2018	9F:74:4E:9F:2B:4D:BA:EC:0F:31:2C:50:B6:56:3B:8E:2D:93:C3:11
cadisigrootr2	21 aprile 2018	B5:61:EB:EA:A4:DE:E4:25:4B:69:1A:98:A5:57:47:C2:34:C7:D9:71
swisssignsilvercag2	21 aprile 2018	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
securetrustca	21 aprile 2018	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:FF:11
cadisigrootr1	21 aprile 2018	8E:1C:74:F8:A6:20:B9:E5:8A:F4:61:FA:EC:2B:47:56:51:1A:52:C6
accvraiz1	21 aprile 2018	93:05:7A:88:15:C6:4F:CE:88:2F:FA:91:16:52:28:78:BC:53:64:17

Nome	Data	SHA1 Impronta digitale
entrustrootcertificationauthority	21 aprile 2018	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
camerfirmaglobalchambersignroot	21 aprile 2018	33:9B:6B:14:50:24:9B:55:7A:01:87:72:84:D9:E0:2F:C3:D2:D8:E9
dstacescax6	21 aprile 2018	40:54:DA:6F:1C:3F:40:74:AC:ED:0F:EC:CD:DB:79:D1:53:FB:90:1D
identrustpublicsectorrootca1	21 aprile 2018	BA:29:41:60:77:98:3F:F4:F3:EF:F2:31:05:3B:2E:EA:6D:4D:45:FD
starfieldrootcertificatesauthorityg2	21 aprile 2018	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
secureglobalca	21 aprile 2018	3A:44:73:5A:E5:81:90:1F:24:86:61:46:1E:3B:9C:C4:5F:F5:3A:1B
eecertificationcenterrootca	21 aprile 2018	C9:A8:B9:E7:55:80:5E:58:E3:53:77:A7:25:EB:AF:C3:7B:27:CC:D7
opentrustrootcag3	21 aprile 2018	6E:26:64:F3:56:BF:34:55:BF:D1:93:3F:7C:01:DE:D8:13:DA:8A:A6
teliasonerarootca1	21 aprile 2018	43:13:BB:96:F1:D5:86:9B:C1:4E:6A:92:F6:CF:F6:34:69:87:82:37
autoridaddecertificacionfirmaprofesionalcifa62634068	21 aprile 2018	AE:C5:FB:3F:C8:E1:BF:C4:E5:4F:03:07:5A:9A:E8:00:B7:F7:B6:FA
opentrustrootcag2	21 aprile 2018	79:5F:88:60:C5:AB:7C:3D:92:E6:CB:F4:8D:E1:45:CD:11:EF:60:0B
opentrustrootcag1	21 aprile 2018	79:91:E8:34:F7:E2:EE:DD:08:95:01:52:E9:55:2D:14:E9:58:D5:7E

Nome	Data	SHA1 Impronta digitale
addtrustexternalroot	21 aprile 2018	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68
chambersofcommerceroot2008	21 aprile 2018	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
digicertglobalrootg3	21 aprile 2018	7E:04:DE:89:6A:3E:66:6D:00:E6:87:D3:3F:FA:D9:3B:E8:3D:34:9E
comodoaaaservicesroot	21 aprile 2018	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
digicertglobalrootg2	21 aprile 2018	DF:3C:24:F9:BF:D6:66:76:1B:26:80:73:FE:06:D1:CC:8D:4F:82:A4
certinomisrootca	21 aprile 2018	9D:70:BB:01:A5:A4:A0:18:11:2E:F7:1C:01:B9:32:C5:34:E7:88:A8
oistewisekeyglobalrootgaca	21 aprile 2018	59:22:A1:E1:5A:EA:16:35:21:F8:98:39:6A:46:46:B0:44:1B:0F:A9
dstrootcax3	21 aprile 2018	DA:C9:02:4F:54:D8:F6:DF:94:93:5F:B1:73:26:38:CA:6A:D7:7C:13
certigna	21 aprile 2018	B1:2E:13:63:45:86:A4:6F:1A:B2:60:68:37:58:2D:C4:AC:FD:94:97
digicerthighassuranceevrootca	21 aprile 2018	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:DC:25
soneraclass2rootca	21 aprile 2018	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
trustcorrootcertca2	21 aprile 2018	B8:BE:6D:CB:56:F1:55:B9:63:D4:12:CA:4E:06:34:C7:94:B2:1C:C0
usertrustrsacertificationauthority	21 aprile 2018	2B:8F:1B:57:33:0D:BB:A2:D0:7A:6C:51:F7:0E:E9:0D:DA:B9:AD:8E

Nome	Data	SHA1 Impronta digitale
trustcorrootcertca1	21 aprile 2018	FF:BD:CD:E7:82:C8:43:5E:3C:6F:26:86:5C:CA:A8:3A:45:5B:C3:0A
geotrustuniversalca	21 aprile 2018	E6:21:F3:35:43:79:05:9A:4B:68:30:9D:8A:2F:74:22:15:87:EC:79
certsignrootca	21 aprile 2018	FA:B7:EE:36:97:26:62:FB:2D:B0:2A:F6:BF:03:FD:E8:7C:4B:2F:9B
amazonrootca4	21 aprile 2018	F6:10:84:07:D6:F8:BB:67:98:0C:C2:E2:44:C2:EB:AE:1C:EF:63:BE
amazonrootca3	21 aprile 2018	0D:44:DD:8C:3C:8C:1A:1A:58:75:64:81:E9:0F:2E:2A:FF:B3:D2:6E
amazonrootca2	21 aprile 2018	5A:8C:EF:45:D7:A6:98:59:76:7A:8C:8B:44:96:B5:78:CF:47:4B:1A
verisignuniversalrootcertificationauthority	21 aprile 2018	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
amazonrootca1	21 aprile 2018	8D:A7:F9:65:EC:5E:FC:37:91:0F:1C:6E:59:FD:C1:CC:6A:6E:DE:16
networksolutionscertificatauthority	21 aprile 2018	74:F8:A3:C3:EF:E7:B3:90:06:4B:83:90:3C:21:64:60:20:E5:DF:CE
thawteprimaryrootca3	21 aprile 2018	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6B:F2
affirmtrustnetworking	21 aprile 2018	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
thawteprimaryrootca2	21 aprile 2018	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:F0:12

Nome	Data	SHA1 Impronta digitale
trustcoreca1	21 aprile 2018	58:D1:DF:95:95:67:6B:63:C0:F0:5B:1C:17:4D:8B:84:0B:C8:78:BD
deutschetelekomrootca2	21 aprile 2018	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD:D6:13:30:FD:8C:DE:37:BF
godaddyrootcertificationauthorityg2	21 aprile 2018	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
entrustrootcertificationauthorityec1	21 aprile 2018	20:D8:06:40:DF:9B:25:F5:12:25:3A:11:EA:F7:59:8A:EB:14:B5:47
szafirrootca2	21 aprile 2018	E2:52:FA:95:3F:ED:DB:24:60:BD:6E:28:F3:9C:CC:CF:5E:B3:3F:DE
tubitakkamusmsslkoksertifिकासिसुरम1	21 aprile 2018	31:43:64:9B:EC:CE:27:EC:ED:3A:3F:0B:8F:0D:E4:E8:91:DD:EE:CA
buypassclass3rootca	21 aprile 2018	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
comodorsacertificationauthority	21 aprile 2018	AF:E5:D2:44:A8:D1:19:42:30:FF:47:9F:E2:F8:97:BB:CD:7A:8C:B4
netlockaranyclassgolderfotanusitvany	21 aprile 2018	06:08:3F:59:3F:15:A1:04:A0:69:A4:6B:A9:03:D0:06:B7:97:09:91
securitycommunicationrootca2	21 aprile 2018	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
dtrustrootclass3ca2ev2009	21 aprile 2018	96:C9:1B:0B:95:B4:10:98:42:FA:D0:D8:22:79:FE:60:FA:B9:16:83
starfieldclass2ca	21 aprile 2018	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:B5:8A

Nome	Data	SHA1 Impronta digitale
pscprocert	21 aprile 2018	70:C1:8D:74:B4:28:81:0A:E4:FD:A5:75:D7:01:9F:99:B0:3D:50:74
actalisauthenticationrootca	21 aprile 2018	F3:73:B3:87:06:5A:28:84:8A:F2:F3:4A:CE:19:2B:DD:C7:8E:9C:AC
staatdernederlandenrootcag3	21 aprile 2018	D8:EB:6B:41:51:92:59:E0:F3:E7:85:00:C0:3D:B6:88:97:C9:EE:FC
cfcae/root	21 aprile 2018	E2:B8:29:4B:55:84:AB:6B:58:C2:90:46:6C:AC:3F:B8:39:8F:84:83
digicerttrustedrootg4	21 aprile 2018	DD:FB:16:CD:49:31:C9:73:A2:03:7D:3F:C8:3A:4D:7D:77:5D:05:E4
staatdernederlandenrootcag2	21 aprile 2018	59:AF:82:79:91:86:C7:B4:75:07:CB:CF:03:57:46:EB:04:DD:B7:16
securitycommunicationevrootca1	21 aprile 2018	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D
globalsignrootcar3	21 aprile 2018	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
globalsignrootcar2	21 aprile 2018	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
certumtrustednetworkkca2	21 aprile 2018	D3:DD:48:3E:2B:BF:4C:05:E8:AF:10:F5:FA:76:26:CF:D3:DC:30:92
acraizfnmtrcm	21 aprile 2018	EC:50:35:07:B2:15:C4:95:62:19:E2:A8:9A:5B:42:99:2C:4C:2C:20
hellenicacademicanresearchinstituteonsecrootca2015	21 aprile 2018	9F:F1:71:8D:92:D5:9A:F3:7D:74:97:B4:BC:6F:84:68:0B:BA:B6:66

Nome	Data	SHA1 Impronta digitale
certplusrootcag2	21 aprile 2018	4F:65:8E:1F:E9:06:D8:28:02:E9:54:47:41:C9:54:25:5D:69:CC:1A
twcarootcertificat ionauthority	21 aprile 2018	CF:9E:87:6D:D3:EB:FC:42:26:97:A3:B5:A3:7A:A0:76:A9:06:23:48
twcaglobalrootca	21 aprile 2018	9C:BB:48:53:F6:A4:F6:D3:52:A4:E8:32:52:55:60:13:F5:AD:AF:65
certplusrootcag1	21 aprile 2018	22:FD:D0:B7:FD:A2:4E:0D:AC:49:2C:A0:AC:A6:7B:6A:1F:E3:F7:66
geotrustuniversalc a2	21 aprile 2018	37:9A:19:7B:41:85:45:35:0C:A6:03:69:F3:3C:2E:AF:47:4F:20:79
baltimorecybertrus troot	21 aprile 2018	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
buypassclass2rootc a	21 aprile 2018	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99
certumtrustednetwo rkca	21 aprile 2018	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28:A2:59:3A:19:A7:0F:06:9E
digicertassuredidr ootg3	21 aprile 2018	F5:17:A2:4F:9A:48:C6:C9:F8:A2:00:26:9F:DC:0F:48:2C:AB:30:89
digicertassuredidr ootg2	21 aprile 2018	A1:4B:48:D9:43:EE:0A:0E:40:90:4F:3C:E0:A4:C0:91:93:51:5D:3F
isrgrootx1	21 aprile 2018	CA:BD:2A:79:A1:07:6A:31:F2:1D:25:36:35:CB:03:9D:43:29:A5:E8
entrustnetpremium2 048secureserverca	21 aprile 2018	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
certplusclass2prim aryca	21 aprile 2018	74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:E2:BB

Nome	Data	SHA1 Impronta digitale
digicertglobalrootca	21 aprile 2018	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:54:36
entrustrootcertificationauthorityg2	21 aprile 2018	8C:F4:27:FD:79:0C:3A:D1:66:06:8D:E8:1E:57:EF:BB:93:22:72:D4
starfieldservicesrootcertificatauthorityg2	21 aprile 2018	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
thawteprimaryrootca	21 aprile 2018	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7B:81
atostrustedroot2011	21 aprile 2018	2B:B1:F5:3E:55:0C:1D:C5:F1:D4:E6:B7:6A:46:4B:55:06:02:AC:21
geotrustglobalca	21 aprile 2018	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:82:12
luxtrustglobalroot2	21 aprile 2018	1E:0E:56:19:0A:D1:8B:25:98:B2:04:44:FF:66:8A:04:17:99:5F:3F
etugracertificatioauthority	21 aprile 2018	51:C6:E7:08:49:06:6E:F3:92:D4:5C:A0:0D:6D:A3:62:8F:C3:52:39
visaecommerceroott	21 aprile 2018	70:17:9B:86:8C:00:A4:FA:60:91:52:22:3F:9F:3E:32:BD:E0:05:62
quovadisrootca	21 aprile 2018	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9
identrustcommercialrootca1	21 aprile 2018	DF:71:7E:AA:4A:D9:4E:C9:55:84:99:60:2D:48:DE:5F:BC:F0:3A:25
staatdernederlandenevrootca	21 aprile 2018	76:E2:7E:C1:4F:DB:82:C1:C0:A6:75:B5:05:BE:3D:29:B4:ED:DB:BB

Nome	Data	SHA1 Impronta digitale
ttelesecglobalrootclass3	21 aprile 2018	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
ttelesecglobalrootclass2	21 aprile 2018	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
comodocertificatio nauthority	21 aprile 2018	66:31:BF:9E:F7:4F:9E:B6:C9:D5:A6:0C:BA:6A:BE:D1:F7:BD:EF:7B
securitycommunicat ionrootca	21 aprile 2018	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
quovadisrootca3g3	21 aprile 2018	48:12:BD:92:3C:A8:C4:39:06:E7:30:6D:27:96:E6:A4:CF:22:2E:7D
xrampglobalcaroot	21 aprile 2018	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
securesignrootca11	21 aprile 2018	3B:C4:9F:48:F8:F3:73:A0:9C:1E:BD:F8:5B:B1:C3:65:C7:D8:11:B3
affirmtrustpremium	21 aprile 2018	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
globalsignrootca	21 aprile 2018	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
swisssigngoldcag2	21 aprile 2018	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
quovadisrootca2g3	21 aprile 2018	09:3C:61:F3:8B:8B:DC:7D:55:DF:75:38:02:05:00:E1:25:F5:C8:36
affirmtrustpremium ecc	21 aprile 2018	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB

Nome	Data	SHA1 Impronta digitale
geotrustprimarycertificatio nauthority	21 aprile 2018	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2: 10:0D:D6:02:90:37:F0:96
quovadisrootca1g3	21 aprile 2018	1B:8E:EA:57:96:29:1A:C9:39:EA:B8:0A: 81:1A:73:73:C0:93:79:67
hongkongpostrootca 1	21 aprile 2018	D6:DA:A8:20:8D:09:D2:15:4D:24:B5:2F: CB:34:6E:B2:58:B2:8A:58
usertrustecccertif icationauthority	21 aprile 2018	D1:CB:CA:5D:B2:D5:2A:7F:69:3B:67:4D: E5:F0:5A:1D:0C:95:7D:F0
cybertrustglobalro ot	21 aprile 2018	5F:43:E5:B1:BF:F8:78:8C:AC:1C:C7:CA: 4A:9A:C6:22:2B:CC:34:C6
godaddyclass2ca	21 aprile 2018	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0: D7:77:70:02:8F:20:EE:E4
hellenicacademican dresearch instituti onsrootca2015	21 aprile 2018	01:0C:06:95:A6:98:19:14:FF:BF:5F:C6: B0:B6:95:EA:29:E9:12:A6
ecacc	21 aprile 2018	28:90:3A:63:5B:52:80:FA:E6:77:4C:0B: 6D:A7:D6:BA:A6:4A:F2:E8
hellenicacademican dresearch instituti onsrootca2011	21 aprile 2018	FE:45:65:9B:79:03:5B:98:A1:61:B5:51: 2E:AC:DA:58:09:48:22:4D
verisignclass3publ icprimary certifica tionauthorityg5	21 aprile 2018	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD: 56:BE:3D:9B:67:44:A5:E5

Nome	Data	SHA1 Impronta digitale
verisignclass3publicprimarycertificationauthorityg4	21 aprile 2018	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
verisignclass3publicprimarycertificationauthorityg3	21 aprile 2018	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
trustisfpsrootca	21 aprile 2018	3B:C0:38:0B:33:C3:F6:A6:0C:86:15:22:93:D9:DF:F5:4B:81:C0:04
epkirootcertificationauthority	21 aprile 2018	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
globalchambersignroot2008	21 aprile 2018	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
camerfirmachambersofcommerceroot	21 aprile 2018	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1
mozillacert81.pem	13 marzo 2014	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28:A2:59:3A:19:A7:0F:06:9E
mozillacert99.pem	13 marzo 2014	F1:7F:6F:B6:31:DC:99:E3:A3:C8:7F:FE:1C:F1:81:10:88:D9:60:33
mozillacert145.pem	13 marzo 2014	10:1D:FA:3F:D5:0B:CB:BB:9B:B5:60:0C:19:55:A4:1A:F4:73:3A:04
mozillacert37.pem	13 marzo 2014	B1:2E:13:63:45:86:A4:6F:1A:B2:60:68:37:58:2D:C4:AC:FD:94:97
mozillacert4.pem	13 marzo 2014	E3:92:51:2F:0A:CF:F5:05:DF:F6:DE:06:7F:75:37:E1:65:EA:57:4B

Nome	Data	SHA1 Impronta digitale
mozillacert70.pem	13 marzo 2014	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50: BA:9E:A8:7E:FE:9A:CE:3C
mozillacert88.pem	13 marzo 2014	FE:45:65:9B:79:03:5B:98:A1:61:B5:51: 2E:AC:DA:58:09:48:22:4D
mozillacert134.pem	13 marzo 2014	70:17:9B:86:8C:00:A4:FA:60:91:52:22: 3F:9F:3E:32:BD:E0:05:62
mozillacert26.pem	13 marzo 2014	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2: 59:3E:7D:44:D9:34:FF:11
mozillacert77.pem	13 marzo 2014	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3: 39:E2:55:76:60:9B:5C:C6
mozillacert123.pem	13 marzo 2014	2A:B6:28:48:5E:78:FB:F3:AD:9E:79:10: DD:6B:DF:99:72:2C:96:E5
mozillacert15.pem	13 marzo 2014	74:20:74:41:72:9C:DD:92:EC:79:31:D8: 23:10:8D:C2:81:92:E2:BB
mozillacert66.pem	13 marzo 2014	DD:E1:D2:A9:01:80:2E:1D:87:5E:84:B3: 80:7E:4B:B1:FD:99:41:34
mozillacert112.pem	13 marzo 2014	43:13:BB:96:F1:D5:86:9B:C1:4E:6A:92: F6:CF:F6:34:69:87:82:37
mozillacert55.pem	13 marzo 2014	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38: DD:F4:1D:DB:08:9E:F0:12
mozillacert101.pem	13 marzo 2014	99:A6:9B:E6:1A:FE:88:6B:4D:2B:82:00: 7C:B8:54:FC:31:7E:15:39
mozillacert119.pem	13 marzo 2014	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F: DD:DE:38:E4:B7:24:2E:FE
mozillacert44.pem	13 marzo 2014	5F:43:E5:B1:BF:F8:78:8C:AC:1C:C7:CA: 4A:9A:C6:22:2B:CC:34:C6

Nome	Data	SHA1 Impronta digitale
mozillacert108.pem	13 marzo 2014	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
mozillacert95.pem	13 marzo 2014	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
mozillacert141.pem	13 marzo 2014	31:7A:2A:D0:7F:2B:33:5E:F5:A1:C3:4E:4B:57:E8:B7:D8:F1:FC:A6
mozillacert33.pem	13 marzo 2014	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D
mozillacert0.pem	13 marzo 2014	97:81:79:50:D8:1C:96:70:CC:34:D8:09:CF:79:44:31:36:7E:F4:74
mozillacert84.pem	13 marzo 2014	D3:C0:63:F2:19:ED:07:3E:34:AD:5D:75:0B:32:76:29:FF:D5:9A:F2
mozillacert130.pem	13 marzo 2014	E5:DF:74:3C:B6:01:C4:9B:98:43:DC:AB:8C:E8:6A:81:10:9F:E4:8E
mozillacert148.pem	13 marzo 2014	04:83:ED:33:99:AC:36:08:05:87:22:ED:BC:5E:46:00:E3:BE:F9:D7
mozillacert22.pem	13 marzo 2014	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
mozillacert7.pem	13 marzo 2014	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:B5:8A
mozillacert73.pem	13 marzo 2014	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
mozillacert137.pem	13 marzo 2014	4A:65:D5:F4:1D:EF:39:B8:B8:90:4A:4A:D3:64:81:33:CF:C7:A1:D1
mozillacert11.pem	13 marzo 2014	05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E:4B:DF:B5:A8:99:B2:4D:43

Nome	Data	SHA1 Impronta digitale
mozillacert29.pem	13 marzo 2014	74:F8:A3:C3:EF:E7:B3:90:06:4B:83:90:3C:21:64:60:20:E5:DF:CE
mozillacert62.pem	13 marzo 2014	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
mozillacert126.pem	13 marzo 2014	25:01:90:19:CF:FB:D9:99:1C:B7:68:25:74:8D:94:5F:30:93:95:42
mozillacert18.pem	13 marzo 2014	79:98:A3:08:E1:4D:65:85:E6:C2:1E:15:3A:71:9F:BA:5A:D3:4A:D9
mozillacert51.pem	13 marzo 2014	FA:B7:EE:36:97:26:62:FB:2D:B0:2A:F6:BF:03:FD:E8:7C:4B:2F:9B
mozillacert69.pem	13 marzo 2014	2F:78:3D:25:52:18:A7:4A:65:39:71:B5:2C:A2:9C:45:15:6F:E9:19
mozillacert115.pem	13 marzo 2014	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
mozillacert40.pem	13 marzo 2014	80:25:EF:F4:6E:70:C8:D4:72:24:65:84:FE:40:3B:8A:8D:6A:DB:F5
mozillacert58.pem	13 marzo 2014	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
mozillacert104.pem	13 marzo 2014	4F:99:AA:93:FB:2B:D1:37:26:A1:99:4A:CE:7F:F0:05:F2:93:5D:1E
mozillacert91.pem	13 marzo 2014	3B:C0:38:0B:33:C3:F6:A6:0C:86:15:22:93:D9:DF:F5:4B:81:C0:04
mozillacert47.pem	13 marzo 2014	1B:4B:39:61:26:27:6B:64:91:A2:68:6D:D7:02:43:21:2D:1F:1D:96
mozillacert80.pem	13 marzo 2014	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB

Nome	Data	SHA1 Impronta digitale
mozillacert98.pem	13 marzo 2014	C9:A8:B9:E7:55:80:5E:58:E3:53:77:A7: 25:EB:AF:C3:7B:27:CC:D7
mozillacert144.pem	13 marzo 2014	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A: B7:41:10:B4:F2:E4:9A:27
mozillacert36.pem	13 marzo 2014	23:88:C9:D3:71:CC:9E:96:3D:FF:7D:3C: A7:CE:FC:D6:25:EC:19:0D
mozillacert3.pem	13 marzo 2014	87:9F:4B:EE:05:DF:98:58:3B:E3:60:D6: 33:E7:0D:3F:FE:98:71:AF
mozillacert87.pem	13 marzo 2014	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC: 19:19:C3:73:34:B9:C7:74
mozillacert133.pem	13 marzo 2014	85:B5:FF:67:9B:0C:79:96:1F:C8:6E:44: 22:00:46:13:DB:17:92:84
mozillacert25.pem	13 marzo 2014	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD: 56:BE:3D:9B:67:44:A5:E5
mozillacert76.pem	13 marzo 2014	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80: DC:E9:6E:2C:C7:B2:78:B7
mozillacert122.pem	13 marzo 2014	02:FA:F3:E2:91:43:54:68:60:78:57:69: 4D:F5:E4:5B:68:85:18:68
mozillacert14.pem	13 marzo 2014	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A: E6:D3:8F:1A:61:C7:DC:25
mozillacert65.pem	13 marzo 2014	69:BD:8C:F4:9C:D3:00:FB:59:2E:17:93: CA:55:6A:F3:EC:AA:35:FB
mozillacert111.pem	13 marzo 2014	9C:BB:48:53:F6:A4:F6:D3:52:A4:E8:32: 52:55:60:13:F5:AD:AF:65
mozillacert129.pem	13 marzo 2014	E6:21:F3:35:43:79:05:9A:4B:68:30:9D: 8A:2F:74:22:15:87:EC:79

Nome	Data	SHA1 Impronta digitale
mozillacert54.pem	13 marzo 2014	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B: 20:D2:D9:32:3A:4C:2A:FD
mozillacert100.pem	13 marzo 2014	58:E8:AB:B0:36:15:33:FB:80:F7:9B:1B: 6D:29:D3:FF:8D:5F:00:F0
mozillacert118.pem	13 marzo 2014	7E:78:4A:10:1C:82:65:CC:2D:E1:F1:6D: 47:B4:40:CA:D9:0A:19:45
mozillacert151.pem	13 marzo 2014	AC:ED:5F:65:53:FD:25:CE:01:5F:1F:7A: 48:3B:6A:74:9F:61:78:C6
mozillacert43.pem	13 marzo 2014	F9:CD:0E:2C:DA:76:24:C1:8F:BD:F0:F0: AB:B6:45:B8:F7:FE:D5:7A
mozillacert107.pem	13 marzo 2014	8E:1C:74:F8:A6:20:B9:E5:8A:F4:61:FA: EC:2B:47:56:51:1A:52:C6
mozillacert94.pem	13 marzo 2014	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6: C7:6B:EB:C6:0B:12:40:99
mozillacert140.pem	13 marzo 2014	CA:3A:FB:CF:12:40:36:4B:44:B2:16:20: 88:80:48:39:19:93:7C:F7
mozillacert32.pem	13 marzo 2014	60:D6:89:74:B5:C2:65:9E:8A:0F:C1:88: 7C:88:D2:46:69:1B:18:2C
mozillacert83.pem	13 marzo 2014	A0:73:E5:C5:BD:43:61:0D:86:4C:21:13: 0A:85:58:57:CC:9C:EA:46
mozillacert147.pem	13 marzo 2014	58:11:9F:0E:12:82:87:EA:50:FD:D9:87: 45:6F:4F:78:DC:FA:D6:D4
mozillacert21.pem	13 marzo 2014	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25: 93:DF:A7:F0:40:D1:1D:CB
mozillacert39.pem	13 marzo 2014	AE:50:83:ED:7C:F4:5C:BC:8F:61:C6:21: FE:68:5D:79:42:21:15:6E

Nome	Data	SHA1 Impronta digitale
mozillacert6.pem	13 marzo 2014	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
mozillacert72.pem	13 marzo 2014	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
mozillacert136.pem	13 marzo 2014	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
mozillacert10.pem	13 marzo 2014	5F:3A:FC:0A:8B:64:F6:86:67:34:74:DF:7E:A9:A2:FE:F9:FA:7A:51
mozillacert28.pem	13 marzo 2014	66:31:BF:9E:F7:4F:9E:B6:C9:D5:A6:0C:BA:6A:BE:D1:F7:BD:EF:7B
mozillacert61.pem	13 marzo 2014	E0:B4:32:2E:B2:F6:A5:68:B6:54:53:84:48:18:4A:50:36:87:43:84
mozillacert79.pem	13 marzo 2014	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
mozillacert125.pem	13 marzo 2014	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
mozillacert17.pem	13 marzo 2014	40:54:DA:6F:1C:3F:40:74:AC:ED:0F:EC:CD:DB:79:D1:53:FB:90:1D
mozillacert50.pem	13 marzo 2014	8C:96:BA:EB:DD:2B:07:07:48:EE:30:32:66:A0:F3:98:6E:7C:AE:58
mozillacert68.pem	13 marzo 2014	AE:C5:FB:3F:C8:E1:BF:C4:E5:4F:03:07:5A:9A:E8:00:B7:F7:B6:FA
mozillacert114.pem	13 marzo 2014	51:C6:E7:08:49:06:6E:F3:92:D4:5C:A0:0D:6D:A3:62:8F:C3:52:39
mozillacert57.pem	13 marzo 2014	D6:DA:A8:20:8D:09:D2:15:4D:24:B5:2F:CB:34:6E:B2:58:B2:8A:58

Nome	Data	SHA1 Impronta digitale
mozillacert103.pem	13 marzo 2014	70:C1:8D:74:B4:28:81:0A:E4:FD:A5:75:D7:01:9F:99:B0:3D:50:74
mozillacert90.pem	13 marzo 2014	F3:73:B3:87:06:5A:28:84:8A:F2:F3:4A:CE:19:2B:DD:C7:8E:9C:AC
mozillacert46.pem	13 marzo 2014	40:9D:4B:D9:17:B5:5C:27:B6:9B:64:CB:98:22:44:0D:CD:09:B8:89
mozillacert97.pem	13 marzo 2014	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F
mozillacert143.pem	13 marzo 2014	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
mozillacert35.pem	13 marzo 2014	2A:C8:D5:8B:57:CE:BF:2F:49:AF:F2:FC:76:8F:51:14:62:90:7A:41
mozillacert2.pem	13 marzo 2014	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
mozillacert86.pem	13 marzo 2014	74:2C:31:92:E6:07:E4:24:EB:45:49:54:2B:E1:BB:C5:3E:61:74:E2
mozillacert132.pem	13 marzo 2014	39:21:C1:15:C1:5D:0E:CA:5C:CB:5B:C4:F0:7D:21:D8:05:0B:56:6A
mozillacert24.pem	13 marzo 2014	59:AF:82:79:91:86:C7:B4:75:07:CB:CF:03:57:46:EB:04:DD:B7:16
mozillacert9.pem	13 marzo 2014	F4:8B:11:BF:DE:AB:BE:94:54:20:71:E6:41:DE:6B:BE:88:2B:40:B9
mozillacert75.pem	13 marzo 2014	D2:32:09:AD:23:D3:14:23:21:74:E4:0D:7F:9D:62:13:97:86:63:3A
mozillacert121.pem	13 marzo 2014	CC:AB:0E:A0:4C:23:01:D6:69:7B:DD:37:9F:CD:12:EB:24:E3:94:9D

Nome	Data	SHA1 Impronta digitale
mozillacert139.pem	13 marzo 2014	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9
mozillacert13.pem	13 marzo 2014	06:08:3F:59:3F:15:A1:04:A0:69:A4:6B:A9:03:D0:06:B7:97:09:91
mozillacert64.pem	13 marzo 2014	62:7F:8D:78:27:65:63:99:D2:7D:7F:90:44:C9:FE:B3:F3:3E:FA:9A
mozillacert110.pem	13 marzo 2014	93:05:7A:88:15:C6:4F:CE:88:2F:FA:91:16:52:28:78:BC:53:64:17
mozillacert128.pem	13 marzo 2014	A9:E9:78:08:14:37:58:88:F2:05:19:B0:6D:2B:0D:2B:60:16:90:7D
mozillacert53.pem	13 marzo 2014	7F:8A:B0:CF:D0:51:87:6A:66:F3:36:0F:47:C8:8D:8C:D3:35:FC:74
mozillacert117.pem	13 marzo 2014	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
mozillacert150.pem	13 marzo 2014	33:9B:6B:14:50:24:9B:55:7A:01:87:72:84:D9:E0:2F:C3:D2:D8:E9
mozillacert42.pem	13 marzo 2014	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD:D6:13:30:FD:8C:DE:37:BF
mozillacert106.pem	13 marzo 2014	E7:A1:90:29:D3:D5:52:DC:0D:0F:C6:92:D3:EA:88:0D:15:2E:1A:6B
mozillacert93.pem	13 marzo 2014	31:F1:FD:68:22:63:20:EE:C6:3B:3F:9D:EA:4A:3E:53:7C:7C:39:17
mozillacert31.pem	13 marzo 2014	9F:74:4E:9F:2B:4D:BA:EC:0F:31:2C:50:B6:56:3B:8E:2D:93:C3:11
mozillacert49.pem	13 marzo 2014	61:57:3A:11:DF:0E:D8:7E:D5:92:65:22:EA:D0:56:D7:44:B3:23:71

Nome	Data	SHA1 Impronta digitale
mozillacert82.pem	13 marzo 2014	2E:14:DA:EC:28:F0:FA:1E:8E:38:9A:4E:AB:EB:26:C0:0A:D3:83:C3
mozillacert146.pem	13 marzo 2014	21:FC:BD:8E:7F:6C:AF:05:1B:D1:B3:43:EC:A8:E7:61:47:F2:0F:8A
mozillacert20.pem	13 marzo 2014	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
mozillacert38.pem	13 marzo 2014	CB:A1:C5:F8:B0:E3:5E:B8:B9:45:12:D3:F9:34:A2:E9:06:10:D3:36
mozillacert5.pem	13 marzo 2014	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
mozillacert71.pem	13 marzo 2014	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
mozillacert89.pem	13 marzo 2014	C8:EC:8C:87:92:69:CB:4B:AB:39:E9:8D:7E:57:67:F3:14:95:73:9D
mozillacert135.pem	13 marzo 2014	62:52:DC:40:F7:11:43:A2:2F:DE:9E:F7:34:8E:06:42:51:B1:81:18
mozillacert27.pem	13 marzo 2014	3A:44:73:5A:E5:81:90:1F:24:86:61:46:1E:3B:9C:C4:5F:F5:3A:1B
mozillacert60.pem	13 marzo 2014	3B:C4:9F:48:F8:F3:73:A0:9C:1E:BD:F8:5B:B1:C3:65:C7:D8:11:B3
mozillacert78.pem	13 marzo 2014	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
mozillacert124.pem	13 marzo 2014	4D:23:78:EC:91:95:39:B5:00:7F:75:8F:03:3B:21:1E:C5:4D:8B:CF
mozillacert16.pem	13 marzo 2014	DA:C9:02:4F:54:D8:F6:DF:94:93:5F:B1:73:26:38:CA:6A:D7:7C:13

Nome	Data	SHA1 Impronta digitale
mozillacert67.pem	13 marzo 2014	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
mozillacert113.pem	13 marzo 2014	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
mozillacert56.pem	13 marzo 2014	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6B:F2
mozillacert102.pem	13 marzo 2014	96:C9:1B:0B:95:B4:10:98:42:FA:D0:D8:22:79:FE:60:FA:B9:16:83
mozillacert45.pem	13 marzo 2014	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
mozillacert109.pem	13 marzo 2014	B5:61:EB:EA:A4:DE:E4:25:4B:69:1A:98:A5:57:47:C2:34:C7:D9:71
mozillacert96.pem	13 marzo 2014	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
mozillacert142.pem	13 marzo 2014	1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:51:85
mozillacert34.pem	13 marzo 2014	59:22:A1:E1:5A:EA:16:35:21:F8:98:39:6A:46:46:B0:44:1B:0F:A9
mozillacert1.pem	13 marzo 2014	23:E5:94:94:51:95:F2:41:48:03:B4:D5:64:D2:A3:A3:F5:D8:8B:8C
mozillacert85.pem	13 marzo 2014	CF:9E:87:6D:D3:EB:FC:42:26:97:A3:B5:A3:7A:A0:76:A9:06:23:48
mozillacert131.pem	13 marzo 2014	37:9A:19:7B:41:85:45:35:0C:A6:03:69:F3:3C:2E:AF:47:4F:20:79
mozillacert149.pem	13 marzo 2014	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1

Nome	Data	SHA1 Impronta digitale
mozillacert23.pem	13 marzo 2014	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2: 99:29:5C:75:6C:81:7B:81
mozillacert8.pem	13 marzo 2014	3E:2B:F7:F2:03:1B:96:F3:8C:E6:C4:D8: A8:5D:3E:2D:58:47:6A:0F
mozillacert74.pem	13 marzo 2014	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A: FF:22:D8:63:E8:25:6F:3F
mozillacert120.pem	13 marzo 2014	DA:40:18:8B:91:89:A3:ED:EE:AE:DA:97: FE:2F:9D:F5:B7:D1:8A:41
mozillacert138.pem	13 marzo 2014	E1:9F:E3:0E:8B:84:60:9E:80:9B:17:0D: 72:A8:C5:BA:6E:14:09:BD
mozillacert12.pem	13 marzo 2014	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D: 40:C6:DD:2F:B1:9C:54:36
mozillacert63.pem	13 marzo 2014	89:DF:74:FE:5C:F4:0F:4A:80:F9:E3:37: 7D:54:DA:91:E1:01:31:8E
mozillacert127.pem	13 marzo 2014	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1: A3:49:A7:F9:96:2A:82:12
mozillacert19.pem	13 marzo 2014	B4:35:D4:E1:11:9D:1C:66:90:A7:49:EB: B3:94:BD:63:7B:A7:82:B7
mozillacert52.pem	13 marzo 2014	8B:AF:4C:9B:1D:F0:2A:92:F7:DA:12:8E: B9:1B:AC:F4:98:60:4B:6F
mozillacert116.pem	13 marzo 2014	2B:B1:F5:3E:55:0C:1D:C5:F1:D4:E6:B7: 6A:46:4B:55:06:02:AC:21
mozillacert41.pem	13 marzo 2014	6B:2F:34:AD:89:58:BE:62:FD:B0:6B:5C: CE:BB:9D:D9:4F:4E:39:F3
mozillacert59.pem	13 marzo 2014	36:79:CA:35:66:87:72:30:4D:30:A5:FB: 87:3B:0F:A7:7B:B7:0D:54

Nome	Data	SHA1 Impronta digitale
mozillacert105.pem	13 marzo 2014	77:47:4F:C6:30:E4:0F:4C:47:64:3F:84:BA:B8:C6:95:4A:8A:41:EC
mozillacert92.pem	13 marzo 2014	A3:F1:33:3F:E2:42:BF:CF:C5:D1:4E:8F:39:42:98:40:68:10:D1:A0
mozillacert30.pem	13 marzo 2014	E7:B4:F6:9D:61:EC:90:69:DB:7E:90:A7:40:1A:3C:F4:7D:4F:E8:EE
mozillacert48.pem	13 marzo 2014	A0:A1:AB:90:C9:FC:84:7B:3B:12:61:E8:97:7D:5F:D3:22:61:D3:CC
verisignc4g2.pem	20 marzo 2014	0B:77:BE:BB:CB:7A:A2:47:05:DE:CC:0F:BD:6A:02:FC:7A:BD:9B:52
verisignc2g3.pem	20 marzo 2014	61:EF:43:D7:7F:CA:D4:61:51:BC:98:E0:C3:59:12:AF:9F:EB:63:11
verisignc1g6.pem	31 dicembre 2014	51:7F:61:1E:29:91:6B:53:82:FB:72:E7:44:D9:8D:C3:CC:53:6D:64
verisignc2g2.pem	20 marzo 2014	B3:EA:C4:47:76:C9:C8:1C:EA:F2:9D:95:B6:CC:A0:08:1B:67:EC:9D
verisignroot.pem	20 marzo 2014	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
verisignc2g1.pem	20 marzo 2014	67:82:AA:E0:ED:EE:E2:1A:58:39:D3:C0:CD:14:68:0A:4F:60:14:2A
verisignc3g5.pem	20 marzo 2014	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
verisignc1g3.pem	20 marzo 2014	20:42:85:DC:F7:EB:76:41:95:57:8E:13:6B:D4:B7:D1:E9:8E:46:A5

Nome	Data	SHA1 Impronta digitale
verisignc3g4.pem	20 marzo 2014	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7: CF:8A:2D:64:C9:3F:6C:3A
verisignc1g2.pem	20 marzo 2014	27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B: 56:16:7F:62:F5:32:E5:47
verisignc3g3.pem	20 marzo 2014	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3: 39:E2:55:76:60:9B:5C:C6
verisignc1g1.pem	20 marzo 2014	90:AE:A2:69:85:FF:14:80:4C:43:49:52: EC:E9:60:84:77:AF:55:6F
verisignc3g2.pem	20 marzo 2014	85:37:1C:A6:E5:50:14:3D:CE:28:03:47: 1B:DE:3A:09:E8:F8:77:0F
verisignc3g1.pem	20 marzo 2014	A1:DB:63:93:91:6F:17:E4:18:55:09:40: 04:15:C7:02:40:B0:AE:6B
verisignc2g6.pem	31 dicembre 2014	40:B3:31:A0:E9:BF:E8:55:BC:39:93:CA: 70:4F:4E:C2:51:D4:1D:8F
verisignc4g3.pem	20 marzo 2014	C8:EC:8C:87:92:69:CB:4B:AB:39:E9:8D: 7E:57:67:F3:14:95:73:9D
gdroot-g2.pem	31 dicembre 2014	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62: A7:9F:45:C2:54:FD:E6:8B
gd-class2-root.pem	31 dicembre 2014	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0: D7:77:70:02:8F:20:EE:E4
gd_bundle-g2.pem	31 dicembre 2014	27:AC:93:69:FA:F2:52:07:BB:26:27:CE: FA:CC:BE:4E:F9:C3:19:B8

Nome	Data	SHA1 Impronta digitale
dstacescax6	18 giugno 2018	40:54:DA:6F:1C:3F:40:74:AC:ED:0F:EC: CD:DB:79:D1:53:FB:90:1D
gd_bundle-g2.pem	18 giugno 2018	27:AC:93:69:FA:F2:52:07:BB:26:27:CE: FA:CC:BE:4E:F9:C3:19:B8
verisignc4g3.pem	18 giugno 2018	C8:EC:8C:87:92:69:CB:4B:AB:39:E9:8D: 7E:57:67:F3:14:95:73:9D
swisssignplatinumg2ca	21 aprile 2018	56:E0:FA:C0:3B:8F:18:23:55:18:E5:D3: 11:CA:E8:C2:43:31:AB:66
geotrustprimarycertificatio nauthorityg3	18 giugno 2018	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B: 20:D2:D9:32:3A:4C:2A:FD
geotrustprimarycertificatio nauthorityg2	18 giugno 2018	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57: 8B:51:9A:99:E6:10:D7:B0
buypassclass2rootca	18 giugno 2018	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6: C7:6B:EB:C6:0B:12:40:99
camerfirmachambersofcommerceroot	18 giugno 2018	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0: DB:72:2E:31:30:61:F0:B1
mozillacert20.pem	18 giugno 2018	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6: 45:25:3A:6F:9F:1A:27:61
mozillacert12.pem	18 giugno 2018	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D: 40:C6:DD:2F:B1:9C:54:36
mozillacert90.pem	18 giugno 2018	F3:73:B3:87:06:5A:28:84:8A:F2:F3:4A: CE:19:2B:DD:C7:8E:9C:AC
mozillacert82.pem	18 giugno 2018	2E:14:DA:EC:28:F0:FA:1E:8E:38:9A:4E: AB:EB:26:C0:0A:D3:83:C3

Nome	Data	SHA1 Impronta digitale
mozillacert140.pem	18 giugno 2018	CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7C:F7
mozillacert74.pem	18 giugno 2018	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
mozillacert132.pem	18 giugno 2018	39:21:C1:15:C1:5D:0E:CA:5C:CB:5B:C4:F0:7D:21:D8:05:0B:56:6A
mozillacert66.pem	18 giugno 2018	DD:E1:D2:A9:01:80:2E:1D:87:5E:84:B3:80:7E:4B:B1:FD:99:41:34
mozillacert124.pem	18 giugno 2018	4D:23:78:EC:91:95:39:B5:00:7F:75:8F:03:3B:21:1E:C5:4D:8B:CF
mozillacert58.pem	18 giugno 2018	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
securitycommunicationrootca2	18 giugno 2018	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
mozillacert116.pem	18 giugno 2018	2B:B1:F5:3E:55:0C:1D:C5:F1:D4:E6:B7:6A:46:4B:55:06:02:AC:21
mozillacert108.pem	18 giugno 2018	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
certigna	18 giugno 2018	B1:2E:13:63:45:86:A4:6F:1A:B2:60:68:37:58:2D:C4:AC:FD:94:97
mozillacert3.pem	18 giugno 2018	87:9F:4B:EE:05:DF:98:58:3B:E3:60:D6:33:E7:0D:3F:FE:98:71:AF
verisignc1g1.pem	18 giugno 2018	90:AE:A2:69:85:FF:14:80:4C:43:49:52:EC:E9:60:84:77:AF:55:6F
verisignc4g2.pem	18 giugno 2018	0B:77:BE:BB:CB:7A:A2:47:05:DE:CC:0F:BD:6A:02:FC:7A:BD:9B:52

Nome	Data	SHA1 Impronta digitale
deutschetelekomrootca2	18 giugno 2018	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD:D6:13:30:FD:8C:DE:37:BF
starfieldrootg2ca	21 aprile 2018	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
comodoecccertificationauthority	18 giugno 2018	9F:74:4E:9F:2B:4D:BA:EC:0F:31:2C:50:B6:56:3B:8E:2D:93:C3:11
digicertglobalrootg3	18 giugno 2018	7E:04:DE:89:6A:3E:66:6D:00:E6:87:D3:3F:FA:D9:3B:E8:3D:34:9E
digicertglobalrootg2	18 giugno 2018	DF:3C:24:F9:BF:D6:66:76:1B:26:80:73:FE:06:D1:CC:8D:4F:82:A4
mozillacert11.pem	18 giugno 2018	05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E:4B:DF:B5:A8:99:B2:4D:43
mozillacert81.pem	18 giugno 2018	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28:A2:59:3A:19:A7:0F:06:9E
mozillacert73.pem	18 giugno 2018	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
szafirrootca2	18 giugno 2018	E2:52:FA:95:3F:ED:DB:24:60:BD:6E:28:F3:9C:CC:CF:5E:B3:3F:DE
mozillacert131.pem	18 giugno 2018	37:9A:19:7B:41:85:45:35:0C:A6:03:69:F3:3C:2E:AF:47:4F:20:79
ecacc	18 giugno 2018	28:90:3A:63:5B:52:80:FA:E6:77:4C:0B:6D:A7:D6:BA:A6:4A:F2:E8
mozillacert65.pem	18 giugno 2018	69:BD:8C:F4:9C:D3:00:FB:59:2E:17:93:CA:55:6A:F3:EC:AA:35:FB

Nome	Data	SHA1 Impronta digitale
turktrustelektroni ksertifik ahizmet sa glayicisih5	18 giugno 2018	C4:18:F6:4D:46:D1:DF:00:3D:27:30:13: 72:43:A9:12:11:C6:75:FB
mozillacert123.pem	18 giugno 2018	2A:B6:28:48:5E:78:FB:F3:AD:9E:79:10: DD:6B:DF:99:72:2C:96:E5
mozillacert57.pem	18 giugno 2018	D6:DA:A8:20:8D:09:D2:15:4D:24:B5:2F: CB:34:6E:B2:58:B2:8A:58
mozillacert115.pem	18 giugno 2018	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62: 32:17:65:CF:17:D8:94:E9
mozillacert49.pem	18 giugno 2018	61:57:3A:11:DF:0E:D8:7E:D5:92:65:22: EA:D0:56:D7:44:B3:23:71
mozillacert107.pem	18 giugno 2018	8E:1C:74:F8:A6:20:B9:E5:8A:F4:61:FA: EC:2B:47:56:51:1A:52:C6
verisignclass3g4ca	21 aprile 2018	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7: CF:8A:2D:64:C9:3F:6C:3A
securetrustca	18 giugno 2018	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2: 59:3E:7D:44:D9:34:FF:11
mozillacert2.pem	18 giugno 2018	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7: CF:8A:2D:64:C9:3F:6C:3A
buypassclass2ca	21 aprile 2018	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6: C7:6B:EB:C6:0B:12:40:99
secomscrootca2	21 aprile 2018	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC: 19:19:C3:73:34:B9:C7:74
secomscrootca1	21 aprile 2018	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38: 0F:C6:56:8F:5D:AC:B2:F7

Nome	Data	SHA1 Impronta digitale
trustisfpsrootca	18 giugno 2018	3B:C0:38:0B:33:C3:F6:A6:0C:86:15:22:93:D9:DF:F5:4B:81:C0:04
hongkongpostrootca1	18 giugno 2018	D6:DA:A8:20:8D:09:D2:15:4D:24:B5:2F:CB:34:6E:B2:58:B2:8A:58
certsignrootca	18 giugno 2018	FA:B7:EE:36:97:26:62:FB:2D:B0:2A:F6:BF:03:FD:E8:7C:4B:2F:9B
geotrustprimaryca	21 aprile 2018	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
twcaglobalrootca	18 giugno 2018	9C:BB:48:53:F6:A4:F6:D3:52:A4:E8:32:52:55:60:13:F5:AD:AF:65
camerfirmachambersca	21 aprile 2018	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
mozillacert10.pem	18 giugno 2018	5F:3A:FC:0A:8B:64:F6:86:67:34:74:DF:7E:A9:A2:FE:F9:FA:7A:51
mozillacert80.pem	18 giugno 2018	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
mozillacert72.pem	18 giugno 2018	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
comodoaaaca	21 aprile 2018	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
mozillacert130.pem	18 giugno 2018	E5:DF:74:3C:B6:01:C4:9B:98:43:DC:AB:8C:E8:6A:81:10:9F:E4:8E
mozillacert64.pem	18 giugno 2018	62:7F:8D:78:27:65:63:99:D2:7D:7F:90:44:C9:FE:B3:F3:3E:FA:9A
mozillacert122.pem	18 giugno 2018	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68

Nome	Data	SHA1 Impronta digitale
mozillacert56.pem	18 giugno 2018	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6B:F2
equifaxsecureebusinessca1	21 aprile 2018	AE:E6:3D:70:E3:76:FB:C7:3A:EB:B0:A1:C1:D4:C4:7A:A7:40:B3:F4
camerfirmachambersignca	21 aprile 2018	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
mozillacert114.pem	18 giugno 2018	51:C6:E7:08:49:06:6E:F3:92:D4:5C:A0:0D:6D:A3:62:8F:C3:52:39
mozillacert48.pem	18 giugno 2018	A0:A1:AB:90:C9:FC:84:7B:3B:12:61:E8:97:7D:5F:D3:22:61:D3:CC
pscprocert	18 giugno 2018	70:C1:8D:74:B4:28:81:0A:E4:FD:A5:75:D7:01:9F:99:B0:3D:50:74
mozillacert106.pem	18 giugno 2018	E7:A1:90:29:D3:D5:52:DC:0D:0F:C6:92:D3:EA:88:0D:15:2E:1A:6B
mozillacert1.pem	18 giugno 2018	23:E5:94:94:51:95:F2:41:48:03:B4:D5:64:D2:A3:A3:F5:D8:8B:8C
eecertificationcenterrootca	18 giugno 2018	C9:A8:B9:E7:55:80:5E:58:E3:53:77:A7:25:EB:AF:C3:7B:27:CC:D7
digicertglobalrootca	18 giugno 2018	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:54:36
thawteprimaryrootcag3	18 giugno 2018	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6B:F2
thawteprimaryrootcag2	18 giugno 2018	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:F0:12

Nome	Data	SHA1 Impronta digitale
entrustrootcertific ationauthorityec1	18 giugno 2018	20:D8:06:40:DF:9B:25:F5:12:25:3A:11: EA:F7:59:8A:EB:14:B5:47
valicertclass2ca	21 aprile 2018	31:7A:2A:D0:7F:2B:33:5E:F5:A1:C3:4E: 4B:57:E8:B7:D8:F1:FC:A6
globalchambersignr oot2008	18 giugno 2018	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52: A1:2C:5B:29:F6:D6:AA:0C
amazonrootca4	18 giugno 2018	F6:10:84:07:D6:F8:BB:67:98:0C:C2:E2: 44:C2:EB:AE:1C:EF:63:BE
gd-class2-root.pem	18 giugno 2018	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0: D7:77:70:02:8F:20:EE:E4
amazonrootca3	18 giugno 2018	0D:44:DD:8C:3C:8C:1A:1A:58:75:64:81: E9:0F:2E:2A:FF:B3:D2:6E
amazonrootca2	18 giugno 2018	5A:8C:EF:45:D7:A6:98:59:76:7A:8C:8B: 44:96:B5:78:CF:47:4B:1A
securitycommunication rootca	18 giugno 2018	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38: 0F:C6:56:8F:5D:AC:B2:F7
amazonrootca1	18 giugno 2018	8D:A7:F9:65:EC:5E:FC:37:91:0F:1C:6E: 59:FD:C1:CC:6A:6E:DE:16
acraizfnmtrcm	18 giugno 2018	EC:50:35:07:B2:15:C4:95:62:19:E2:A8: 9A:5B:42:99:2C:4C:2C:20
quovadisrootca3g3	18 giugno 2018	48:12:BD:92:3C:A8:C4:39:06:E7:30:6D: 27:96:E6:A4:CF:22:2E:7D
certplusrootcag2	18 giugno 2018	4F:65:8E:1F:E9:06:D8:28:02:E9:54:47: 41:C9:54:25:5D:69:CC:1A

Nome	Data	SHA1 Impronta digitale
certplusrootcag1	18 giugno 2018	22:FD:D0:B7:FD:A2:4E:0D:AC:49:2C:A0: AC:A6:7B:6A:1F:E3:F7:66
mozillacert71.pem	18 giugno 2018	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52: A1:2C:5B:29:F6:D6:AA:0C
mozillacert63.pem	18 giugno 2018	89:DF:74:FE:5C:F4:0F:4A:80:F9:E3:37: 7D:54:DA:91:E1:01:31:8E
mozillacert121.pem	18 giugno 2018	CC:AB:0E:A0:4C:23:01:D6:69:7B:DD:37: 9F:CD:12:EB:24:E3:94:9D
ttelesecglobalroot class3ca	21 aprile 2018	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70: 19:9D:2A:BE:11:E3:81:D1
mozillacert55.pem	18 giugno 2018	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38: DD:F4:1D:DB:08:9E:F0:12
mozillacert113.pem	18 giugno 2018	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB: E7:92:7D:7D:65:2D:34:31
baltimorecybertrust tca	21 aprile 2018	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88: 2C:78:DB:28:52:CA:E4:74
mozillacert47.pem	18 giugno 2018	1B:4B:39:61:26:27:6B:64:91:A2:68:6D: D7:02:43:21:2D:1F:1D:96
mozillacert105.pem	18 giugno 2018	77:47:4F:C6:30:E4:0F:4C:47:64:3F:84: BA:B8:C6:95:4A:8A:41:EC
mozillacert39.pem	18 giugno 2018	AE:50:83:ED:7C:F4:5C:BC:8F:61:C6:21: FE:68:5D:79:42:21:15:6E
usertrustecccertif icationauthority	18 giugno 2018	D1:CB:CA:5D:B2:D5:2A:7F:69:3B:67:4D: E5:F0:5A:1D:0C:95:7D:F0
mozillacert0.pem	18 giugno 2018	97:81:79:50:D8:1C:96:70:CC:34:D8:09: CF:79:44:31:36:7E:F4:74

Nome	Data	SHA1 Impronta digitale
securitycommunicationevrootca1	18 giugno 2018	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D
verisignc3g5.pem	18 giugno 2018	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
globalsignr3ca	21 aprile 2018	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
trustcoreca1	18 giugno 2018	58:D1:DF:95:95:67:6B:63:C0:F0:5B:1C:17:4D:8B:84:0B:C8:78:BD
equifaxsecureglobalbusinessca1	21 aprile 2018	3A:74:CB:7A:47:DB:70:DE:89:1F:24:35:98:64:B8:2D:82:BD:1A:36
geotrustuniversalca	18 giugno 2018	E6:21:F3:35:43:79:05:9A:4B:68:30:9D:8A:2F:74:22:15:87:EC:79
affirmtrustpremiumca	21 aprile 2018	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
staatdernederlandenrootcag3	18 giugno 2018	D8:EB:6B:41:51:92:59:E0:F3:E7:85:00:C0:3D:B6:88:97:C9:EE:FC
staatdernederlandenrootcag2	18 giugno 2018	59:AF:82:79:91:86:C7:B4:75:07:CB:CF:03:57:46:EB:04:DD:B7:16
mozillacert70.pem	18 giugno 2018	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
secomevrootca1	21 aprile 2018	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D
geotrustglobalca	18 giugno 2018	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:82:12
mozillacert62.pem	18 giugno 2018	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B

Nome	Data	SHA1 Impronta digitale
mozillacert120.pem	18 giugno 2018	DA:40:18:8B:91:89:A3:ED:EE:AE:DA:97:FE:2F:9D:F5:B7:D1:8A:41
mozillacert54.pem	18 giugno 2018	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
mozillacert112.pem	18 giugno 2018	43:13:BB:96:F1:D5:86:9B:C1:4E:6A:92:F6:CF:F6:34:69:87:82:37
mozillacert46.pem	18 giugno 2018	40:9D:4B:D9:17:B5:5C:27:B6:9B:64:CB:98:22:44:0D:CD:09:B8:89
swisssigngoldcag2	18 giugno 2018	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
mozillacert104.pem	18 giugno 2018	4F:99:AA:93:FB:2B:D1:37:26:A1:99:4A:CE:7F:F0:05:F2:93:5D:1E
mozillacert38.pem	18 giugno 2018	CB:A1:C5:F8:B0:E3:5E:B8:B9:45:12:D3:F9:34:A2:E9:06:10:D3:36
certplusclass3ppri maryca	21 aprile 2018	21:6B:2A:29:E6:2A:00:CE:82:01:46:D8:24:41:41:B9:25:11:B2:79
entrustrootcertifi cationauthorityg2	18 giugno 2018	8C:F4:27:FD:79:0C:3A:D1:66:06:8D:E8:1E:57:EF:BB:93:22:72:D4
godaddyrootg2ca	21 aprile 2018	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
cfcaevroot	18 giugno 2018	E2:B8:29:4B:55:84:AB:6B:58:C2:90:46:6C:AC:3F:B8:39:8F:84:83
verisignc3g4.pem	18 giugno 2018	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
geotrustuniversalc a2	18 giugno 2018	37:9A:19:7B:41:85:45:35:0C:A6:03:69:F3:3C:2E:AF:47:4F:20:79

Nome	Data	SHA1 Impronta digitale
starfieldservicesrootg2ca	21 aprile 2018	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
digicerthighassuranceevrootca	18 giugno 2018	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:DC:25
entrustnetpremium2048secureserverca	18 giugno 2018	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
camerfirmaglobalchambersignroot	18 giugno 2018	33:9B:6B:14:50:24:9B:55:7A:01:87:72:84:D9:E0:2F:C3:D2:D8:E9
verisignclass3g3ca	21 aprile 2018	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
godaddyclass2ca	18 giugno 2018	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
mozillacert61.pem	18 giugno 2018	E0:B4:32:2E:B2:F6:A5:68:B6:54:53:84:48:18:4A:50:36:87:43:84
mozillacert53.pem	18 giugno 2018	7F:8A:B0:CF:D0:51:87:6A:66:F3:36:0F:47:C8:8D:8C:D3:35:FC:74
atostrustedroot2011	18 giugno 2018	2B:B1:F5:3E:55:0C:1D:C5:F1:D4:E6:B7:6A:46:4B:55:06:02:AC:21
mozillacert111.pem	18 giugno 2018	9C:BB:48:53:F6:A4:F6:D3:52:A4:E8:32:52:55:60:13:F5:AD:AF:65
staatdernederlandenevrootca	18 giugno 2018	76:E2:7E:C1:4F:DB:82:C1:C0:A6:75:B5:05:BE:3D:29:B4:ED:DB:BB
mozillacert45.pem	18 giugno 2018	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
mozillacert103.pem	18 giugno 2018	70:C1:8D:74:B4:28:81:0A:E4:FD:A5:75:D7:01:9F:99:B0:3D:50:74

Nome	Data	SHA1 Impronta digitale
mozillacert37.pem	18 giugno 2018	B1:2E:13:63:45:86:A4:6F:1A:B2:60:68:37:58:2D:C4:AC:FD:94:97
mozillacert29.pem	18 giugno 2018	74:F8:A3:C3:EF:E7:B3:90:06:4B:83:90:3C:21:64:60:20:E5:DF:CE
izenpecom	18 giugno 2018	2F:78:3D:25:52:18:A7:4A:65:39:71:B5:2C:A2:9C:45:15:6F:E9:19
comodorsacertificationauthority	18 giugno 2018	AF:E5:D2:44:A8:D1:19:42:30:FF:47:9F:E2:F8:97:BB:CD:7A:8C:B4
mozillacert99.pem	18 giugno 2018	F1:7F:6F:B6:31:DC:99:E3:A3:C8:7F:FE:1C:F1:81:10:88:D9:60:33
mozillacert149.pem	18 giugno 2018	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1
utnuserfirstobjectca	21 aprile 2018	E1:2D:FB:4B:41:D7:D9:C3:2B:30:51:4B:AC:1D:81:D8:38:5E:2D:46
verisignc3g3.pem	18 giugno 2018	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
dstrootcax3	18 giugno 2018	DA:C9:02:4F:54:D8:F6:DF:94:93:5F:B1:73:26:38:CA:6A:D7:7C:13
addtrustexternalroot	18 giugno 2018	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68
certumtrustednetworkca	18 giugno 2018	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28:A2:59:3A:19:A7:0F:06:9E
affirmtrustpremiumecc	18 giugno 2018	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
starfieldclass2ca	18 giugno 2018	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:B5:8A

Nome	Data	SHA1 Impronta digitale
actalisauthenticationrootca	18 giugno 2018	F3:73:B3:87:06:5A:28:84:8A:F2:F3:4A:CE:19:2B:DD:C7:8E:9C:AC
verisignclass2g3ca	21 aprile 2018	61:EF:43:D7:7F:CA:D4:61:51:BC:98:E0:C3:59:12:AF:9F:EB:63:11
isrgrootx1	18 giugno 2018	CA:BD:2A:79:A1:07:6A:31:F2:1D:25:36:35:CB:03:9D:43:29:A5:E8
godaddyrootcertificateauthorityg2	18 giugno 2018	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
mozillacert60.pem	18 giugno 2018	3B:C4:9F:48:F8:F3:73:A0:9C:1E:BD:F8:5B:B1:C3:65:C7:D8:11:B3
chunghwaepkirootca	21 aprile 2018	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
mozillacert52.pem	18 giugno 2018	8B:AF:4C:9B:1D:F0:2A:92:F7:DA:12:8E:B9:1B:AC:F4:98:60:4B:6F
microseceszignorootca2009	18 giugno 2018	89:DF:74:FE:5C:F4:0F:4A:80:F9:E3:37:7D:54:DA:91:E1:01:31:8E
securesignrootca11	18 giugno 2018	3B:C4:9F:48:F8:F3:73:A0:9C:1E:BD:F8:5B:B1:C3:65:C7:D8:11:B3
mozillacert110.pem	18 giugno 2018	93:05:7A:88:15:C6:4F:CE:88:2F:FA:91:16:52:28:78:BC:53:64:17
mozillacert44.pem	18 giugno 2018	5F:43:E5:B1:BF:F8:78:8C:AC:1C:C7:CA:4A:9A:C6:22:2B:CC:34:C6
mozillacert102.pem	18 giugno 2018	96:C9:1B:0B:95:B4:10:98:42:FA:D0:D8:22:79:FE:60:FA:B9:16:83
mozillacert36.pem	18 giugno 2018	23:88:C9:D3:71:CC:9E:96:3D:FF:7D:3C:A7:CE:FC:D6:25:EC:19:0D

Nome	Data	SHA1 Impronta digitale
mozillacert28.pem	18 giugno 2018	66:31:BF:9E:F7:4F:9E:B6:C9:D5:A6:0C: BA:6A:BE:D1:F7:BD:EF:7B
baltimorecybertrustroot	18 giugno 2018	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88: 2C:78:DB:28:52:CA:E4:74
amzninternalrootca	12 dicembre 2008	A7:B7:F6:15:8A:FF:1E:C8:85:13:38:BC: 93:EB:A2:AB:A4:09:EF:06
mozillacert98.pem	18 giugno 2018	C9:A8:B9:E7:55:80:5E:58:E3:53:77:A7: 25:EB:AF:C3:7B:27:CC:D7
mozillacert148.pem	18 giugno 2018	04:83:ED:33:99:AC:36:08:05:87:22:ED: BC:5E:46:00:E3:BE:F9:D7
verisignc3g2.pem	18 giugno 2018	85:37:1C:A6:E5:50:14:3D:CE:28:03:47: 1B:DE:3A:09:E8:F8:77:0F
quovadisrootca2g3	18 giugno 2018	09:3C:61:F3:8B:8B:DC:7D:55:DF:75:38: 02:05:00:E1:25:F5:C8:36
geotrustprimarycertificatio nauthority	18 giugno 2018	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2: 10:0D:D6:02:90:37:F0:96
opentrustrootcag3	18 giugno 2018	6E:26:64:F3:56:BF:34:55:BF:D1:93:3F: 7C:01:DE:D8:13:DA:8A:A6
opentrustrootcag2	18 giugno 2018	79:5F:88:60:C5:AB:7C:3D:92:E6:CB:F4: 8D:E1:45:CD:11:EF:60:0B
opentrustrootcag1	18 giugno 2018	79:91:E8:34:F7:E2:EE:DD:08:95:01:52: E9:55:2D:14:E9:58:D5:7E
verisignclass3ca	21 aprile 2018	A1:DB:63:93:91:6F:17:E4:18:55:09:40: 04:15:C7:02:40:B0:AE:6B

Nome	Data	SHA1 Impronta digitale
globalsignca	21 aprile 2018	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
ttelesecglobalrootclass2ca	21 aprile 2018	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
verisignclass1g3ca	21 aprile 2018	20:42:85:DC:F7:EB:76:41:95:57:8E:13:6B:D4:B7:D1:E9:8E:46:A5
verisignuniversalrootca	21 aprile 2018	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
soneraclass2ca	21 aprile 2018	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
starfieldservicesrootcertificateauthorityg2	18 giugno 2018	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
mozillacert51.pem	18 giugno 2018	FA:B7:EE:36:97:26:62:FB:2D:B0:2A:F6:BF:03:FD:E8:7C:4B:2F:9B
mozillacert43.pem	18 giugno 2018	F9:CD:0E:2C:DA:76:24:C1:8F:BD:F0:F0:AB:B6:45:B8:F7:FE:D5:7A
mozillacert101.pem	18 giugno 2018	99:A6:9B:E6:1A:FE:88:6B:4D:2B:82:00:7C:B8:54:FC:31:7E:15:39
mozillacert35.pem	18 giugno 2018	2A:C8:D5:8B:57:CE:BF:2F:49:AF:F2:FC:76:8F:51:14:62:90:7A:41
globalsignr2ca	21 aprile 2018	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
mozillacert27.pem	18 giugno 2018	3A:44:73:5A:E5:81:90:1F:24:86:61:46:1E:3B:9C:C4:5F:F5:3A:1B

Nome	Data	SHA1 Impronta digitale
affirmtrustpremium	18 giugno 2018	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
mozillacert19.pem	18 giugno 2018	B4:35:D4:E1:11:9D:1C:66:90:A7:49:EB:B3:94:BD:63:7B:A7:82:B7
mozillacert97.pem	18 giugno 2018	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F
netlockaranyclassg oldfotanusitvany	18 giugno 2018	06:08:3F:59:3F:15:A1:04:A0:69:A4:6B:A9:03:D0:06:B7:97:09:91
mozillacert89.pem	18 giugno 2018	C8:EC:8C:87:92:69:CB:4B:AB:39:E9:8D:7E:57:67:F3:14:95:73:9D
verisignroot.pem	18 giugno 2018	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
mozillacert147.pem	18 giugno 2018	58:11:9F:0E:12:82:87:EA:50:FD:D9:87:45:6F:4F:78:DC:FA:D6:D4
aolrootca2	21 aprile 2018	85:B5:FF:67:9B:0C:79:96:1F:C8:6E:44:22:00:46:13:DB:17:92:84
cia-crt-g3-01-ca	23 novembre 2016	2B:EE:2C:BA:A3:1D:B5:FE:60:40:41:95:08:ED:46:82:39:4D:ED:E2
aolrootca1	21 aprile 2018	39:21:C1:15:C1:5D:0E:CA:5C:CB:5B:C4:F0:7D:21:D8:05:0B:56:6A
verisignc3g1.pem	18 giugno 2018	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
mozillacert139.pem	18 giugno 2018	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9

Nome	Data	SHA1 Impronta digitale
soneraclass2rootca	18 giugno 2018	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A: B7:41:10:B4:F2:E4:9A:27
swisssignsilverg2ca	21 aprile 2018	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25: 93:DF:A7:F0:40:D1:1D:CB
thawteprimaryrootca	18 giugno 2018	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2: 99:29:5C:75:6C:81:7B:81
gdcatrustauthr5root	18 giugno 2018	0F:36:38:5B:81:1A:25:C3:9B:31:4E:83: CA:E9:34:66:70:CC:74:B4
trustcenterclass4caii	21 aprile 2018	A6:9A:91:FD:05:7F:13:6A:42:63:0B:B1: 76:0D:2D:51:12:0C:16:50
usertrustrsacertificationauthority	18 giugno 2018	2B:8F:1B:57:33:0D:BB:A2:D0:7A:6C:51: F7:0E:E9:0D:DA:B9:AD:8E
digicertassuredidrootg3	18 giugno 2018	F5:17:A2:4F:9A:48:C6:C9:F8:A2:00:26: 9F:DC:0F:48:2C:AB:30:89
digicertassuredidrootg2	18 giugno 2018	A1:4B:48:D9:43:EE:0A:0E:40:90:4F:3C: E0:A4:C0:91:93:51:5D:3F
mozillacert50.pem	18 giugno 2018	8C:96:BA:EB:DD:2B:07:07:48:EE:30:32: 66:A0:F3:98:6E:7C:AE:58
mozillacert42.pem	18 giugno 2018	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD: D6:13:30:FD:8C:DE:37:BF
mozillacert100.pem	18 giugno 2018	58:E8:AB:B0:36:15:33:FB:80:F7:9B:1B: 6D:29:D3:FF:8D:5F:00:F0
mozillacert34.pem	18 giugno 2018	59:22:A1:E1:5A:EA:16:35:21:F8:98:39: 6A:46:46:B0:44:1B:0F:A9
affirmtrustcommercialca	21 aprile 2018	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80: DC:E9:6E:2C:C7:B2:78:B7

Nome	Data	SHA1 Impronta digitale
mozillacert26.pem	18 giugno 2018	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:FF:11
globalsigneccrootc ar5	18 giugno 2018	1F:24:C6:30:CD:A4:18:EF:20:69:FF:AD:4F:DD:5F:46:3A:1B:69:AA
globalsigneccrootc ar4	18 giugno 2018	69:69:56:2E:40:80:F4:24:A1:E7:19:9F:14:BA:F3:EE:58:AB:6A:BB
buypassclass3rootc a	18 giugno 2018	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
mozillacert18.pem	18 giugno 2018	79:98:A3:08:E1:4D:65:85:E6:C2:1E:15:3A:71:9F:BA:5A:D3:4A:D9
mozillacert96.pem	18 giugno 2018	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
verisignc2g6.pem	18 giugno 2018	40:B3:31:A0:E9:BF:E8:55:BC:39:93:CA:70:4F:4E:C2:51:D4:1D:8F
secomvalicertclass 1ca	21 aprile 2018	E5:DF:74:3C:B6:01:C4:9B:98:43:DC:AB:8C:E8:6A:81:10:9F:E4:8E
mozillacert88.pem	18 giugno 2018	FE:45:65:9B:79:03:5B:98:A1:61:B5:51:2E:AC:DA:58:09:48:22:4D
accvraiz1	18 giugno 2018	93:05:7A:88:15:C6:4F:CE:88:2F:FA:91:16:52:28:78:BC:53:64:17
mozillacert146.pem	18 giugno 2018	21:FC:BD:8E:7F:6C:AF:05:1B:D1:B3:43:EC:A8:E7:61:47:F2:0F:8A
mozillacert138.pem	18 giugno 2018	E1:9F:E3:0E:8B:84:60:9E:80:9B:17:0D:72:A8:C5:BA:6E:14:09:BD
verisignclass3g2ca	21 aprile 2018	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F

Nome	Data	SHA1 Impronta digitale
dtrustrootclass3ca2ev2009	18 giugno 2018	96:C9:1B:0B:95:B4:10:98:42:FA:D0:D8:22:79:FE:60:FA:B9:16:83
xrampglobalca	21 aprile 2018	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
mozillacert9.pem	18 giugno 2018	F4:8B:11:BF:DE:AB:BE:94:54:20:71:E6:41:DE:6B:BE:88:2B:40:B9
verisignuniversalrootcertificationauthority	18 giugno 2018	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
tubitakkamusmsslkoksertifिकासिसुरम1	18 giugno 2018	31:43:64:9B:EC:CE:27:EC:ED:3A:3F:0B:8F:0D:E4:E8:91:DD:EE:CA
mozillacert41.pem	18 giugno 2018	6B:2F:34:AD:89:58:BE:62:FD:B0:6B:5C:CE:BB:9D:D9:4F:4E:39:F3
mozillacert33.pem	18 giugno 2018	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D
mozillacert25.pem	18 giugno 2018	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
mozillacert17.pem	18 giugno 2018	40:54:DA:6F:1C:3F:40:74:AC:ED:0F:EC:CD:DB:79:D1:53:FB:90:1D
mozillacert95.pem	18 giugno 2018	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
affirmtrustpremiumeccca	21 aprile 2018	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
mozillacert87.pem	18 giugno 2018	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74

Nome	Data	SHA1 Impronta digitale
mozillacert145.pem	18 giugno 2018	10:1D:FA:3F:D5:0B:CB:BB:9B:B5:60:0C: 19:55:A4:1A:F4:73:3A:04
mozillacert79.pem	18 giugno 2018	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F: 7D:6A:06:65:26:32:28:27
mozillacert137.pem	18 giugno 2018	4A:65:D5:F4:1D:EF:39:B8:B8:90:4A:4A: D3:64:81:33:CF:C7:A1:D1
digicertassuredidr ootca	18 giugno 2018	05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E: 4B:DF:B5:A8:99:B2:4D:43
addtrustqualifiedc a	21 aprile 2018	4D:23:78:EC:91:95:39:B5:00:7F:75:8F: 03:3B:21:1E:C5:4D:8B:CF
mozillacert129.pem	18 giugno 2018	E6:21:F3:35:43:79:05:9A:4B:68:30:9D: 8A:2F:74:22:15:87:EC:79
verisignclass2g2ca	21 aprile 2018	B3:EA:C4:47:76:C9:C8:1C:EA:F2:9D:95: B6:CC:A0:08:1B:67:EC:9D
baltimorecodesigni ngca	21 aprile 2018	30:46:D8:C8:88:FF:69:30:C3:4A:FC:CD: 49:27:08:7C:60:56:7B:0D
luxtrustglobalroot 2	18 giugno 2018	1E:0E:56:19:0A:D1:8B:25:98:B2:04:44: FF:66:8A:04:17:99:5F:3F
visaecommercerooot	18 giugno 2018	70:17:9B:86:8C:00:A4:FA:60:91:52:22: 3F:9F:3E:32:BD:E0:05:62
oistewisekeyglobal rootgbca	18 giugno 2018	0F:F9:40:76:18:D3:D7:6A:4B:98:F0:A8: 35:9E:0C:FD:27:AC:CC:ED
mozillacert8.pem	18 giugno 2018	3E:2B:F7:F2:03:1B:96:F3:8C:E6:C4:D8: A8:5D:3E:2D:58:47:6A:0F
comodocertificatio nauthority	18 giugno 2018	66:31:BF:9E:F7:4F:9E:B6:C9:D5:A6:0C: BA:6A:BE:D1:F7:BD:EF:7B

Nome	Data	SHA1 Impronta digitale
cia-crt-g3-02-ca	23 novembre 2016	96:4A:BB:A7:BD:DA:FC:97:34:C0:0A:2D:F0:05:98:F7:E6:C6:6F:09
verisignc1g6.pem	18 giugno 2018	51:7F:61:1E:29:91:6B:53:82:FB:72:E7:44:D9:8D:C3:CC:53:6D:64
trustcenterclass2caii	21 aprile 2018	AE:50:83:ED:7C:F4:5C:BC:8F:61:C6:21:FE:68:5D:79:42:21:15:6E
quovadisrootca1g3	18 giugno 2018	1B:8E:EA:57:96:29:1A:C9:39:EA:B8:0A:81:1A:73:73:C0:93:79:67
mozillacert40.pem	18 giugno 2018	80:25:EF:F4:6E:70:C8:D4:72:24:65:84:FE:40:3B:8A:8D:6A:DB:F5
cadisigrootr2	18 giugno 2018	B5:61:EB:EA:A4:DE:E4:25:4B:69:1A:98:A5:57:47:C2:34:C7:D9:71
cadisigrootr1	18 giugno 2018	8E:1C:74:F8:A6:20:B9:E5:8A:F4:61:FA:EC:2B:47:56:51:1A:52:C6
mozillacert32.pem	18 giugno 2018	60:D6:89:74:B5:C2:65:9E:8A:0F:C1:88:7C:88:D2:46:69:1B:18:2C
utndatacorpsgccca	21 aprile 2018	58:11:9F:0E:12:82:87:EA:50:FD:D9:87:45:6F:4F:78:DC:FA:D6:D4
mozillacert24.pem	18 giugno 2018	59:AF:82:79:91:86:C7:B4:75:07:CB:CF:03:57:46:EB:04:DD:B7:16
addtrustclass1ca	21 aprile 2018	CC:AB:0E:A0:4C:23:01:D6:69:7B:DD:37:9F:CD:12:EB:24:E3:94:9D
mozillacert16.pem	18 giugno 2018	DA:C9:02:4F:54:D8:F6:DF:94:93:5F:B1:73:26:38:CA:6A:D7:7C:13

Nome	Data	SHA1 Impronta digitale
affirmtrustnetworkingca	21 aprile 2018	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
mozillacert94.pem	18 giugno 2018	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99
mozillacert86.pem	18 giugno 2018	74:2C:31:92:E6:07:E4:24:EB:45:49:54:2B:E1:BB:C5:3E:61:74:E2
mozillacert144.pem	18 giugno 2018	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
mozillacert78.pem	18 giugno 2018	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
mozillacert136.pem	18 giugno 2018	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
mozillacert128.pem	18 giugno 2018	A9:E9:78:08:14:37:58:88:F2:05:19:B0:6D:2B:0D:2B:60:16:90:7D
verisignclass1g2ca	21 aprile 2018	27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:62:F5:32:E5:47
hellenicacademican dresearch instituti onsrootca2015	18 giugno 2018	01:0C:06:95:A6:98:19:14:FF:BF:5F:C6:B0:B6:95:EA:29:E9:12:A6
soneraclass1ca	21 aprile 2018	07:47:22:01:99:CE:74:B9:7C:B0:3D:79:B2:64:A2:C8:55:E9:33:FF
hellenicacademican dresearch instituti onsrootca2011	18 giugno 2018	FE:45:65:9B:79:03:5B:98:A1:61:B5:51:2E:AC:DA:58:09:48:22:4D

Nome	Data	SHA1 Impronta digitale
certumtrustednetworkrkca2	18 giugno 2018	D3:DD:48:3E:2B:BF:4C:05:E8:AF:10:F5:FA:76:26:CF:D3:DC:30:92
equifaxsecureca	21 aprile 2018	D2:32:09:AD:23:D3:14:23:21:74:E4:0D:7F:9D:62:13:97:86:63:3A
thawteserverca	21 aprile 2018	9F:AD:91:A6:CE:6A:C6:C5:00:47:C4:4E:C9:D4:A5:0D:92:D8:49:79
mozillacert7.pem	18 giugno 2018	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:B5:8A
affirmtrustnetworking	18 giugno 2018	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
deprecateditsecca	27 gennaio 2012	12:12:0B:03:0E:15:14:54:F4:DD:B3:F5:DE:13:6E:83:5A:29:72:9D
globalsignrootcar3	18 giugno 2018	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
globalsignrootcar2	18 giugno 2018	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
quovadisrootca	18 giugno 2018	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9
mozillacert31.pem	18 giugno 2018	9F:74:4E:9F:2B:4D:BA:EC:0F:31:2C:50:B6:56:3B:8E:2D:93:C3:11
entrustrootcertificationauthority	18 giugno 2018	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
mozillacert23.pem	18 giugno 2018	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7B:81
mozillacert15.pem	18 giugno 2018	74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:E2:BB

Nome	Data	SHA1 Impronta digitale
verisignc2g3.pem	18 giugno 2018	61:EF:43:D7:7F:CA:D4:61:51:BC:98:E0:C3:59:12:AF:9F:EB:63:11
mozillacert93.pem	18 giugno 2018	31:F1:FD:68:22:63:20:EE:C6:3B:3F:9D:EA:4A:3E:53:7C:7C:39:17
mozillacert151.pem	18 giugno 2018	AC:ED:5F:65:53:FD:25:CE:01:5F:1F:7A:48:3B:6A:74:9F:61:78:C6
mozillacert85.pem	18 giugno 2018	CF:9E:87:6D:D3:EB:FC:42:26:97:A3:B5:A3:7A:A0:76:A9:06:23:48
certplusclass2primaryca	18 giugno 2018	74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:E2:BB
mozillacert143.pem	18 giugno 2018	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
mozillacert77.pem	18 giugno 2018	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
mozillacert135.pem	18 giugno 2018	62:52:DC:40:F7:11:43:A2:2F:DE:9E:F7:34:8E:06:42:51:B1:81:18
mozillacert69.pem	18 giugno 2018	2F:78:3D:25:52:18:A7:4A:65:39:71:B5:2C:A2:9C:45:15:6F:E9:19
mozillacert127.pem	18 giugno 2018	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:82:12
mozillacert119.pem	18 giugno 2018	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
geotrustprimaryca3	21 aprile 2018	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
identrustpublicsectorrootca1	18 giugno 2018	BA:29:41:60:77:98:3F:F4:F3:EF:F2:31:05:3B:2E:EA:6D:4D:45:FD

Nome	Data	SHA1 Impronta digitale
geotrustprimarycag2	21 aprile 2018	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
trustcorrootcertca2	18 giugno 2018	B8:BE:6D:CB:56:F1:55:B9:63:D4:12:CA:4E:06:34:C7:94:B2:1C:C0
mozillacert6.pem	18 giugno 2018	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
trustcorrootcertca1	18 giugno 2018	FF:BD:CD:E7:82:C8:43:5E:3C:6F:26:86:5C:CA:A8:3A:45:5B:C3:0A
networksolutionscertificatauthority	18 giugno 2018	74:F8:A3:C3:EF:E7:B3:90:06:4B:83:90:3C:21:64:60:20:E5:DF:CE
twcarootcertificationauthority	18 giugno 2018	CF:9E:87:6D:D3:EB:FC:42:26:97:A3:B5:A3:7A:A0:76:A9:06:23:48
addtrustexternalca	21 aprile 2018	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68
verisignclass3g5ca	21 aprile 2018	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
autoridaddecertificacionfirmaprofesi onalcifa62634068	18 giugno 2018	AE:C5:FB:3F:C8:E1:BF:C4:E5:4F:03:07:5A:9A:E8:00:B7:F7:B6:FA
hellenicacademicanresearchinstitutesonsecrootca2015	18 giugno 2018	9F:F1:71:8D:92:D5:9A:F3:7D:74:97:B4:BC:6F:84:68:0B:BA:B6:66
verisigntsaca	21 aprile 2018	20:CE:B1:F0:F5:1C:0E:19:A9:F3:8D:B1:AA:8E:03:8C:AA:7A:C7:01

Nome	Data	SHA1 Impronta digitale
utnuserfirsthardwa reca	21 aprile 2018	04:83:ED:33:99:AC:36:08:05:87:22:ED: BC:5E:46:00:E3:BE:F9:D7
identrustcommercia lrootca1	18 giugno 2018	DF:71:7E:AA:4A:D9:4E:C9:55:84:99:60: 2D:48:DE:5F:BC:F0:3A:25
dtrustrootclass3ca 22009	18 giugno 2018	58:E8:AB:B0:36:15:33:FB:80:F7:9B:1B: 6D:29:D3:FF:8D:5F:00:F0
epkirootcertificat ionauthority	18 giugno 2018	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4: 56:4B:CF:E2:3D:69:C6:F0
mozillacert30.pem	18 giugno 2018	E7:B4:F6:9D:61:EC:90:69:DB:7E:90:A7: 40:1A:3C:F4:7D:4F:E8:EE
teliasonerarootcav 1	18 giugno 2018	43:13:BB:96:F1:D5:86:9B:C1:4E:6A:92: F6:CF:F6:34:69:87:82:37
buypassclass3ca	21 aprile 2018	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD: C7:C2:81:A5:BC:A9:64:57
mozillacert22.pem	18 giugno 2018	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2: 10:0D:D6:02:90:37:F0:96
mozillacert14.pem	18 giugno 2018	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A: E6:D3:8F:1A:61:C7:DC:25
verisignc2g2.pem	18 giugno 2018	B3:EA:C4:47:76:C9:C8:1C:EA:F2:9D:95: B6:CC:A0:08:1B:67:EC:9D
certumca	21 aprile 2018	62:52:DC:40:F7:11:43:A2:2F:DE:9E:F7: 34:8E:06:42:51:B1:81:18
mozillacert92.pem	18 giugno 2018	A3:F1:33:3F:E2:42:BF:CF:C5:D1:4E:8F: 39:42:98:40:68:10:D1:A0
mozillacert150.pem	18 giugno 2018	33:9B:6B:14:50:24:9B:55:7A:01:87:72: 84:D9:E0:2F:C3:D2:D8:E9

Nome	Data	SHA1 Impronta digitale
mozillacert84.pem	18 giugno 2018	D3:C0:63:F2:19:ED:07:3E:34:AD:5D:75:0B:32:76:29:FF:D5:9A:F2
ttelesecglobalrootclass3	18 giugno 2018	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
globalsignrootca	18 giugno 2018	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
ttelesecglobalrootclass2	18 giugno 2018	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
mozillacert142.pem	18 giugno 2018	1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:51:85
mozillacert76.pem	18 giugno 2018	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
mozillacert134.pem	18 giugno 2018	70:17:9B:86:8C:00:A4:FA:60:91:52:22:3F:9F:3E:32:BD:E0:05:62
mozillacert68.pem	18 giugno 2018	AE:C5:FB:3F:C8:E1:BF:C4:E5:4F:03:07:5A:9A:E8:00:B7:F7:B6:FA
etugracertificatio nauthority	18 giugno 2018	51:C6:E7:08:49:06:6E:F3:92:D4:5C:A0:0D:6D:A3:62:8F:C3:52:39
mozillacert126.pem	18 giugno 2018	25:01:90:19:CF:FB:D9:99:1C:B7:68:25:74:8D:94:5F:30:93:95:42
keynectisrootca	21 aprile 2018	9C:61:5C:4D:4D:85:10:3A:53:26:C2:4D:BA:EA:E4:A2:D2:D5:CC:97
mozillacert118.pem	18 giugno 2018	7E:78:4A:10:1C:82:65:CC:2D:E1:F1:6D:47:B4:40:CA:D9:0A:19:45
quovadisrootca3	18 giugno 2018	1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:51:85

Nome	Data	SHA1 Impronta digitale
quovadisrootca2	18 giugno 2018	CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7C:F7
mozillacert5.pem	18 giugno 2018	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
verisignc1g3.pem	18 giugno 2018	20:42:85:DC:F7:EB:76:41:95:57:8E:13:6B:D4:B7:D1:E9:8E:46:A5
cybertrustglobalroot	18 giugno 2018	5F:43:E5:B1:BF:F8:78:8C:AC:1C:C7:CA:4A:9A:C6:22:2B:CC:34:C6
amzninternalinfosecag3	27 febbraio 2015	B9:B1:CA:38:F7:BF:9C:D2:D4:95:E7:B6:5E:75:32:9B:A8:78:2E:F6
starfieldrootcertificateauthorityg2	18 giugno 2018	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
entrust2048ca	21 aprile 2018	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
swisssignsilverca2	18 giugno 2018	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
affirmtrustcommercial	18 giugno 2018	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
certinomisrootca	18 giugno 2018	9D:70:BB:01:A5:A4:A0:18:11:2E:F7:1C:01:B9:32:C5:34:E7:88:A8
xrampglobalcaroot	18 giugno 2018	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
secureglobalca	18 giugno 2018	3A:44:73:5A:E5:81:90:1F:24:86:61:46:1E:3B:9C:C4:5F:F5:3A:1B
swisssigngoldg2ca	21 aprile 2018	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61

Nome	Data	SHA1 Impronta digitale
mozillacert21.pem	18 giugno 2018	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
mozillacert13.pem	18 giugno 2018	06:08:3F:59:3F:15:A1:04:A0:69:A4:6B:A9:03:D0:06:B7:97:09:91
verisignc2g1.pem	18 giugno 2018	67:82:AA:E0:ED:EE:E2:1A:58:39:D3:C0:CD:14:68:0A:4F:60:14:2A
mozillacert91.pem	18 giugno 2018	3B:C0:38:0B:33:C3:F6:A6:0C:86:15:22:93:D9:DF:F5:4B:81:C0:04
oistewisekeyglobal rootgaca	18 giugno 2018	59:22:A1:E1:5A:EA:16:35:21:F8:98:39:6A:46:46:B0:44:1B:0F:A9
mozillacert83.pem	18 giugno 2018	A0:73:E5:C5:BD:43:61:0D:86:4C:21:13:0A:85:58:57:CC:9C:EA:46
entrustevca	21 aprile 2018	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
mozillacert141.pem	18 giugno 2018	31:7A:2A:D0:7F:2B:33:5E:F5:A1:C3:4E:4B:57:E8:B7:D8:F1:FC:A6
mozillacert75.pem	18 giugno 2018	D2:32:09:AD:23:D3:14:23:21:74:E4:0D:7F:9D:62:13:97:86:63:3A
mozillacert133.pem	18 giugno 2018	85:B5:FF:67:9B:0C:79:96:1F:C8:6E:44:22:00:46:13:DB:17:92:84
mozillacert67.pem	18 giugno 2018	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
mozillacert125.pem	18 giugno 2018	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
mozillacert59.pem	18 giugno 2018	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54

Nome	Data	SHA1 Impronta digitale
thawtepremiumserve rca	21 aprile 2018	E0:AB:05:94:20:72:54:93:05:60:62:02: 36:70:F7:CD:2E:FC:66:66
mozillacert117.pem	18 giugno 2018	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88: 2C:78:DB:28:52:CA:E4:74
utnuserfirstclient authemailca	21 aprile 2018	B1:72:B1:A5:6D:95:F9:1F:E5:02:87:E1: 4D:37:EA:6A:44:63:76:8A
entrustrootcag2	21 aprile 2018	8C:F4:27:FD:79:0C:3A:D1:66:06:8D:E8: 1E:57:EF:BB:93:22:72:D4
mozillacert109.pem	18 giugno 2018	B5:61:EB:EA:A4:DE:E4:25:4B:69:1A:98: A5:57:47:C2:34:C7:D9:71
digicerttrustedroo tg4	18 giugno 2018	DD:FB:16:CD:49:31:C9:73:A2:03:7D:3F: C8:3A:4D:7D:77:5D:05:E4
gdroot-g2.pem	18 giugno 2018	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62: A7:9F:45:C2:54:FD:E6:8B
comodoaaaservicesr oot	18 giugno 2018	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2: F1:F1:60:17:64:D8:E3:49
mozillacert4.pem	18 giugno 2018	E3:92:51:2F:0A:CF:F5:05:DF:F6:DE:06: 7F:75:37:E1:65:EA:57:4B
verisignclass3publ icprimary certifica tionauthorityg5	18 giugno 2018	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD: 56:BE:3D:9B:67:44:A5:E5
chambersofcommerce root2008	18 giugno 2018	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50: BA:9E:A8:7E:FE:9A:CE:3C

Nome	Data	SHA1 Impronta digitale
verisignclass3publicprimarycertificationauthorityg4	18 giugno 2018	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
verisignclass3publicprimarycertificationauthorityg3	18 giugno 2018	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
thawtepersonalfreemailca	21 aprile 2018	E6:18:83:AE:84:CA:C1:C1:CD:52:AD:E8:E9:25:2B:45:A6:4F:B7:E2
verisignc1g2.pem	18 giugno 2018	27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:62:F5:32:E5:47
gtecybertrustglobalca	21 aprile 2018	97:81:79:50:D8:1C:96:70:CC:34:D8:09:CF:79:44:31:36:7E:F4:74
trustcenteruniversalcai	21 aprile 2018	6B:2F:34:AD:89:58:BE:62:FD:B0:6B:5C:CE:BB:9D:D9:4F:4E:39:F3
camerfirmachamberscommerceca	21 aprile 2018	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1
verisignclass1ca	21 aprile 2018	CE:6A:64:A3:09:E4:2F:BB:D9:85:1C:45:3E:64:09:EA:E8:7D:60:F1

AWS AppSync log delle modifiche del modello di mappatura del resolver

Note

Ora supportiamo principalmente il runtime APPSYNC_JS e la relativa documentazione.

[Prendi in considerazione l'utilizzo del runtime APPSYNC_JS e delle relative guide qui.](#)

I modelli di mappatura del resolver e della funzione prevedono versioni multiple. La versione del modello di mappatura, ad esempio), impone quanto segue 2018-05-29:

- La forma prevista della configurazione della richiesta di origine dati fornita dal modello di richiesta
- Il comportamento di esecuzione del modello di mappatura della richiesta e del modello di mappatura della risposta

Le versioni sono rappresentate utilizzando il YYYY-MM-DD formato, una data successiva corrisponde a una versione più recente. Questa pagina elenca le differenze tra le versioni dei modelli di mappatura attualmente supportate in AWS AppSync.

Argomenti

- [Disponibilità delle operazioni dell'origine dati per matrice di versione](#)
- [Modifica della versione su un modello di mappatura del resolver di unità.](#)
- [Modifica della versione su una funzione](#)
- [2018-05-29](#)
- [2017-02-28](#)

Disponibilità delle operazioni dell'origine dati per matrice di versione

Operazione/versionsupportata	2017-02-28	2018-05-29
AWS Lambda Invoca	Sì	Sì
AWS Lambda BatchInvoke	Sì	Sì
Nessuna origine dati	Sì	Sì
Amazon OpenSearch GET	Sì	Sì
Amazon OpenSearch POST	Sì	Sì
Amazon OpenSearch PUT	Sì	Sì
Amazon OpenSearch ELIMINA	Sì	Sì

Operazione/versione supportata	2017-02-28	2018-05-29
Amazon OpenSearch GET	Sì	Sì
DynamoDB GetItem	Sì	Sì
Scan di DynamoDB	Sì	Sì
Query di DynamoDB	Sì	Sì
DynamoDB DeleteItem	Sì	Sì
DynamoDB PutItem	Sì	Sì
DynamoDB BatchGetItem	No	Sì
DynamoDB BatchPutItem	No	Sì
DynamoDB BatchDeleteItem	No	Sì
HTTP	No	Sì
Amazon RDS	No	Sì

Nota: solo la versione 2018-05-29 è attualmente supportata nelle funzioni.

Modifica della versione su un modello di mappatura del resolver di unità.

Per i resolver di unità, la versione viene specificata come parte del corpo del modello di mappatura della richiesta. Per aggiornare la versione, è sufficiente aggiornare il campo `version` alla nuova versione.

Ad esempio, per aggiornare la versione del modello: AWS Lambda

```
{
  "version": "2017-02-28",
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "arguments": $utils.toJson($context.arguments)
  }
}
```

```
}  
}
```

È necessario aggiornare il campo dalla versione 2017-02-28 alla 2018-05-29 nel seguente modo:

```
{  
  "version": "2018-05-29", ## Note the version  
  "operation": "Invoke",  
  "payload": {  
    "field": "getPost",  
    "arguments": $utils.toJson($context.arguments)  
  }  
}
```

Modifica della versione su una funzione

Per le funzioni, la versione è specificata come campo `functionVersion` nell'oggetto della funzione. Per aggiornare la versione, basta aggiornare la `functionVersion`. Nota: al momento, solo la versione 2018-05-29 è supportata per la funzione.

Di seguito è riportato un esempio di un comando dell'interfaccia a riga di comando per aggiornare la versione di una funzione esistente:

```
aws appsync update-function \  
--api-id REPLACE_WITH_API_ID \  
--function-id REPLACE_WITH_FUNCTION_ID \  
--data-source-name "PostTable" \  
--function-version "2018-05-29" \  
--request-mapping-template "{...}" \  
--response-mapping-template "$util.toJson($ctx.result)"
```

Nota: si consiglia di omettere il campo della versione dal modello di mappatura della richiesta di funzione, perché non verrà rispettato. Se non specifichi una versione all'interno del modello di mappatura della richiesta di funzione, il valore della versione verrà sostituito da quello nel campo `functionVersion`.

2018-05-29

Modifica del comportamento

- Se il risultato dell'invocazione dell'origine dati è `null`, viene eseguito il modello di mappatura della risposta.
- Se l'invocazione dell'origine dati genera un errore, tocca a te gestirlo, il risultato valutato del modello di mappatura della risposta sarà sempre inserito all'interno del blocco di `data` della risposta GraphQL.

Ragionamento

- Un risultato `null` di un'invocazione ha un significato e in alcuni casi d'uso è preferibile gestire i risultati `null` in modo personalizzato. Ad esempio, un'applicazione potrebbe verificare se esiste un record in una tabella Amazon DynamoDB per eseguire alcuni controlli di autorizzazione. In tal caso, un risultato `null` dell'invocazione vorrebbe dire che l'utente potrebbe non essere autorizzato. L'esecuzione del modello di mappatura della risposta offre ora la possibilità di generare un errore non autorizzato. Questo comportamento offre un maggiore controllo al progettista API.

Dato il seguente modello di mappatura della risposta:

```
$util.toJson($ctx.result)
```

In passato, nella versione 2017-02-28, se `$ctx.result` tornava `null`, il modello di mappatura della risposta non veniva eseguito. Con la versione 2018-05-29, ora possiamo gestire questo scenario. Ad esempio, è possibile scegliere di generare un errore di autorizzazione nel modo seguente:

```
# throw an unauthorized error if the result is null
#if ( $util.isNull($ctx.result) )
    $util.unauthorized()
#end
$util.toJson($ctx.result)
```

Nota: a volte gli errori restituiti da un'origine dati non sono fatali né previsti, ecco perché il modello di mappatura della risposta deve avere la flessibilità di gestire l'errore di invocazione e decidere se ignorarlo, rigenerarlo o generarne un altro.

Dato il seguente modello di mappatura della risposta:

```
$util.toJson($ctx.result)
```

In passato, nella versione 2017-02-28, in caso di un errore di invocazione, veniva valutato il modello di mappatura della risposta e il risultato veniva inserito automaticamente nel blocco `errors` della risposta GraphQL. Con la versione 2018-05-29, è ora possibile scegliere cosa fare con l'errore, se rigenerarlo, generarne uno diverso o aggiungere l'errore durante la restituzione dei dati.

Rigenerare un errore di invocazione

Nel seguente modello di risposta, generiamo lo stesso errore restituito dall'origine dati.

```
#if ( $ctx.error )
    $util.error($ctx.error.message, $ctx.error.type)
#end
$util.toJson($ctx.result)
```

Nel caso di un errore di invocazione (ad esempio se è presente `$ctx.error`) la risposta è simile alla seguente:

```
{
  "data": {
    "getPost": null
  },
  "errors": [
    {
      "path": [
        "getPost"
      ],
      "errorType": "DynamoDB:ConditionalCheckFailedException",
      "message": "Conditional check failed exception...",
      "locations": [
        {
          "line": 5,
          "column": 5
        }
      ]
    }
  ]
}
```

Generare un errore diverso

Nel seguente modello di risposta, generiamo lo stesso errore personalizzato dopo aver elaborato l'errore restituito dall'origine dati.

```
#if ( $ctx.error )
  #if ( $ctx.error.type.equals("ConditionalCheckFailedException") )
    ## we choose here to change the type and message of the error for
    ConditionalCheckFailedExceptions
    $util.error("Error while updating the post, try again. Error:
    $ctx.error.message", "UpdateError")
  #else
    $util.error($ctx.error.message, $ctx.error.type)
  #end
#end
$util.toJson($ctx.result)
```

Nel caso di un errore di invocazione (ad esempio se è presente `$ctx.error`) la risposta è simile alla seguente:

```
{
  "data": {
    "getPost": null
  },
  "errors": [
    {
      "path": [
        "getPost"
      ],
      "errorType": "UpdateError",
      "message": "Error while updating the post, try again. Error: Conditional
check failed exception...",
      "locations": [
        {
          "line": 5,
          "column": 5
        }
      ]
    }
  ]
}
```

Aggiungere un errore ai dati restituiti

Nel seguente modello di risposta, aggiungiamo lo stesso errore restituito dall'origine dati mentre reinseriamo i dati all'interno della risposta. Questo è noto anche come risposta parziale.

```
#if ( $ctx.error )
  $util.appendError($ctx.error.message, $ctx.error.type)
  #set($defaultPost = {id: "1", title: 'default post'})
  $util.toJson($defaultPost)
#else
  $util.toJson($ctx.result)
#end
```

Nel caso di un errore di invocazione (ad esempio se è presente `$ctx.error`) la risposta è simile alla seguente:

```
{
  "data": {
    "getPost": {
      "id": "1",
      "title": "A post"
    }
  },
  "errors": [
    {
      "path": [
        "getPost"
      ],
      "errorType": "ConditionalCheckFailedException",
      "message": "Conditional check failed exception...",
      "locations": [
        {
          "line": 5,
          "column": 5
        }
      ]
    }
  ]
}
```


Migrazione dalla versione 2017-02-28 alla 2018-05-29

La migrazione dalla versione 2017-02-28 alla 2018-05-29 è semplice. Modifica il campo della versione nel modello di mappatura della richiesta del resolver o nell'oggetto della versione della funzione. Tuttavia, tieni presente che l'esecuzione della versione 2018-05-29 si comporta in modo diverso dalla 2017-02-28; le modifiche sono evidenziate [qui](#).

Mantenere lo stesso comportamento di esecuzione dalla versione 2017-02-28 alla 2018-05-29

In alcuni casi, è possibile mantenere lo stesso comportamento di esecuzione della versione 2017-02-28 durante l'esecuzione di un modello che ha la versione 2018-05-29.

Esempio: DynamoDB PutItem

Dato il seguente modello di richiesta DynamoDB PutItem del 28/02/2017:

```
{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key": {
    "foo" : ... typed value,
    "bar" : ... typed value
  },
  "attributeValues" : {
    "baz" : ... typed value
  },
  "condition" : {
    ...
  }
}
```

E il seguente modello di risposta:

```
$util.toJson($ctx.result)
```

La migrazione alla versione 2018-05-29 modifica questi modelli nel modo seguente:

```
{
  "version" : "2018-05-29", ## Note the new 2018-05-29 version
  "operation" : "PutItem",
  "key": {
```

```

    "foo" : ... typed value,
    "bar" : ... typed value
  },
  "attributeValues" : {
    "baz" : ... typed value
  },
  "condition" : {
    ...
  }
}

```

E modifica il modello di risposta nel modo seguente:

```

## If there is a datasource invocation error, we choose to raise the same error
## the field data will be set to null.
#if($ctx.error)
  $util.error($ctx.error.message, $ctx.error.type, $ctx.result)
#end

## If the data source invocation is null, we return null.
#if($util.isNull($ctx.result))
  #return
#end

$util.toJson($ctx.result)

```

Ora che hai la responsabilità di gestire gli errori, abbiamo scelto di generare lo stesso errore utilizzando `$util.error()` che è stato restituito da DynamoDB. Puoi adattare questo frammento per convertire il modello di mappatura alla versione 2018-05-29, ma devi ricordare che se il modello di risposta è diverso dovrai tenere conto delle modifiche al comportamento di esecuzione.

Esempio: DynamoDB GetItem

Dato il seguente modello di richiesta DynamoDB GetItem del 28/02/2017:

```

{
  "version" : "2017-02-28",
  "operation" : "GetItem",
  "key" : {
    "foo" : ... typed value,
    "bar" : ... typed value
  },

```

```
"consistentRead" : true
}
```

E il seguente modello di risposta:

```
## map table attribute postId to field Post.id
$util.qr($ctx.result.put("id", $ctx.result.get("postId")))

$util.toJson($ctx.result)
```

La migrazione alla versione 2018-05-29 modifica questi modelli nel modo seguente:

```
{
  "version" : "2018-05-29", ## Note the new 2018-05-29 version
  "operation" : "GetItem",
  "key" : {
    "foo" : ... typed value,
    "bar" : ... typed value
  },
  "consistentRead" : true
}
```

E modifica il modello di risposta nel modo seguente:

```
## If there is a datasource invocation error, we choose to raise the same error
#if($ctx.error)
  $util.error($ctx.error.message, $ctx.error.type)
#end

## If the data source invocation is null, we return null.
#if($util.isNull($ctx.result))
  #return
#end

## map table attribute postId to field Post.id
$util.qr($ctx.result.put("id", $ctx.result.get("postId")))

$util.toJson($ctx.result)
```

Nella versione 2017-02-28, se l'invocazione dell'origine dati era null, cioè non c'era alcuna voce nella tabella DynamoDB che corrispondeva alla nostra chiave, il modello di mappatura della risposta

non sarebbe stato eseguito. Questo poteva andare bene nella maggior parte dei casi, ma se prevedevi che `$ctx.result` non fosse `null`, questo scenario deve essere gestito.

2017-02-28

Caratteristiche

- Se il risultato dell'invocazione dell'origine dati è `null`, il modello di mappatura della risposta non viene eseguito.
- Se l'invocazione dell'origine dati genera un errore, viene eseguito il modello di mappatura della risposta e il risultato valutato viene inserito all'interno del blocco di `errors.data` della risposta GraphQL.

Riferimento al tipo GraphQL

I tipi scalari in GraphQL rappresentano valori leaf primitivi in uno schema GraphQL. Questi sono i tipi di dati più basilari che si risolvono in un unico valore. A differenza dei tipi di oggetti, i tipi scalari non possono avere sottocampi. GraphQL viene fornito con un set di tipi scalari predefiniti:

- Int: un numero intero con segno a 32 bit
- Float: un valore a virgola mobile con segno a doppia precisione
- String: una sequenza di caratteri UTF-8
- Boolean: un valore vero o falso
- ID: un identificatore univoco, spesso usato per recuperare un oggetto o come chiave per una cache

Questi tipi scalari fungono da elementi costitutivi per i tipi più complessi dello schema. Vengono utilizzati per definire campi che contengono valori semplici e singolari. Oltre a questi scalari integrati, AWS AppSync fornisce scalari aggiuntivi per diversi casi d'uso.

Le interfacce e le unioni in GraphQL sono tipi astratti che consentono una progettazione di schemi flessibile ed estensibile. Forniscono meccanismi per raggruppare i tipi correlati e abilitare le query polimorfe. Un'interfaccia in GraphQL è un tipo astratto che definisce un insieme di campi che un tipo deve includere per implementare l'interfaccia. Funge da contratto per gli oggetti specificando un insieme comune di campi che i tipi di implementazione devono avere. Le interfacce sono utili quando si desidera restituire un oggetto o un campo che può essere di diversi tipi, ma che ha comunque alcuni campi garantiti. Al contrario, un Union in GraphQL rappresenta un tipo che potrebbe essere uno dei diversi tipi di oggetto, ma non definisce alcun campo comune tra questi tipi. Le unioni sono utili quando è necessario restituire un campo che può essere di più tipi e questi tipi non condividono necessariamente campi comuni. Sia le interfacce che le unioni sono particolarmente utili negli scenari in cui un campo può restituire diversi tipi di dati, consentendo ai client di eseguire query per campi specifici in base al tipo restituito.

Questa sezione viene utilizzata come riferimento per i tipi di schema.

Argomenti

- [Tipi scalari in GraphQL](#)
- [Interfacce e unioni in GraphQL](#)

Tipi scalari in GraphQL

Un tipo di oggetto GraphQL ha un nome e campi e tali campi possono avere sottocampi. In definitiva, i campi di un tipo di oggetto devono risolversi in tipi scalari, che rappresentano i fogli della query. Per ulteriori informazioni sui tipi di oggetti e sugli scalari, consulta [Schemi e tipi sul sito](#) Web GraphQL.

Oltre al set predefinito di scalari GraphQL, consente AWS AppSync anche di utilizzare gli scalari definiti dal servizio che iniziano con il prefisso. AWS AppSync non supporta la creazione di scalari definiti dall'utente (personalizzati). È necessario utilizzare gli scalari predefiniti o AWS quelli scalari.

Non è possibile utilizzare AWS come prefisso per tipi di oggetti personalizzati.

La sezione seguente è un riferimento per la digitazione dello schema.

Scalari predefiniti

GraphQL definisce i seguenti scalari predefiniti:

Elenco degli scalari predefiniti

ID

Un identificatore univoco per un oggetto. Questo scalare è serializzato come un `String` ma non è pensato per essere leggibile dall'uomo.

String

Una sequenza di caratteri UTF-8.

Int

Un valore intero compreso tra $-(2^{31})$ e $2^{31}-1$.

Float

Un valore in virgola mobile IEEE 754.

Boolean

Un valore booleano, `true` o `false`.

AWS AppSync scalari

AWS AppSync definisce i seguenti scalari:

AWS AppSync elenco degli scalari

AWSDDate

Una stringa di [data ISO 8601](#) estesa nel formato. YYYY-MM-DD

AWSTime

Una stringa [temporale ISO 8601](#) estesa nel formato. hh:mm:ss.sss

AWSDatetime

Una stringa estesa di [data e ora ISO 8601](#) nel formato. YYYY-MM-DDThh:mm:ss.sssZ

Note

Gli AWSDatetime scalari AWSDDateAWSTime, e possono facoltativamente includere un offset del fuso [orario](#). Ad esempio, i valori 1970-01-01Z1970-01-01-07:00, e 1970-01-01+05:30 sono tutti validi per. AWSDDate L'offset del fuso orario deve essere Z (UTC) o un offset in ore e minuti (e, facoltativamente, secondi). Ad esempio ±hh:mm:ss. Il campo dei secondi nell'offset del fuso orario è considerato valido anche se non fa parte dello standard ISO 8601.

AWSTimestamp

Un valore intero che rappresenta il numero di secondi prima o dopo. 1970-01-01-T00:00Z

AWSEmail

Un indirizzo e-mail nel formato local-part@domain-part definito da [RFC 822](#).

AWSJSON

Una stringa JSON. Qualsiasi costrutto JSON valido viene automaticamente analizzato e caricato nel codice del resolver come mappe, elenchi o valori scalari anziché come stringhe di input letterali. Le stringhe senza virgolette o il codice JSON altrimenti non valido generano un errore di convalida GraphQL.

AWSPhone

Un numero di telefono. Questo valore viene memorizzato come stringa. I numeri di telefono possono contenere spazi o trattini per separare gruppi di cifre. Si presume che i numeri di telefono

senza prefisso internazionale siano numeri statunitensi o nordamericani che aderiscono al North American Numbering Plan ([NANP](#)).

AWSURL

[Un URL come definito da RFC 1738](#). Ad esempio, `https://www.amazon.com/dp/B000NZW3KC/` o `mailto:example@example.com` URLs deve contenere uno schema (`http`, `mailto`) e non può contenere due barre (`//`) nella parte del percorso.

AWSIPAddress

Un IPv6 indirizzo IPv4 o valido. IPv4 gli indirizzi sono previsti in notazione quadrupla (). `123.12.34.56` IPv6 gli indirizzi sono previsti in formato non tra parentesi e separati da due punti (). `1a2b:3c4b::1234:4567` È possibile includere un suffisso CIDR opzionale () `123.45.67.89/16` per indicare la subnet mask.

Esempio di utilizzo dello schema

Lo schema GraphQL di esempio seguente utilizza tutti gli scalari personalizzati come «oggetto» e mostra i modelli di richiesta e risposta del resolver per le operazioni di base di `put`, `get` e `list`. Infine, l'esempio mostra come è possibile utilizzarlo durante l'esecuzione di query e mutazioni.

```
type Mutation {
  putObject(
    email: AWSEmail,
    json: AWSJSON,
    date: AWSDate,
    time: AWSTime,
    datetime: AWSDateTime,
    timestamp: AWSTimestamp,
    url: AWSURL,
    phoneno: AWSPhone,
    ip: AWSIPAddress
  ): Object
}

type Object {
  id: ID!
  email: AWSEmail
  json: AWSJSON
  date: AWSDate
  time: AWSTime
```



```

    datetime: AWSDateTime
    timestamp: AWSTimestamp
    url: AWSURL
    phoneno: AWSPhone
    ip: AWSIPAddress
  }

  type Query {
    getObject(id: ID!): Object
    listObjects: [Object]
  }

  schema {
    query: Query
    mutation: Mutation
  }

```

Ecco come `putObject` potrebbe apparire un modello di richiesta. A `putObject` utilizza un'`PutItem` operazione per creare o aggiornare un elemento nella tabella Amazon DynamoDB. Tieni presente che questo frammento di codice non ha una tabella Amazon DynamoDB configurata come origine dati. Questo viene usato solo come esempio:

```

{
  "version" : "2017-02-28",
  "operation" : "PutItem",
  "key" : {
    "id": $util.dynamodb.toDynamoDBJson($util.autoId()),
  },
  "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}

```

Il modello di risposta per `putObject` restituisce i risultati:

```
$util.toJson($ctx.result)
```

Ecco come `getObject` potrebbe apparire un modello di richiesta. A `getObject` utilizza un'`GetItem` operazione per restituire un insieme di attributi per l'elemento a cui è stata assegnata la chiave primaria. Tieni presente che questo frammento di codice non ha una tabella Amazon DynamoDB configurata come origine dati. Questo viene usato solo come esempio:

```
{
```

```

    "version": "2017-02-28",
    "operation": "GetItem",
    "key": {
      "id": $util.dynamodb.toDynamoDBJson($ctx.args.id),
    }
  }
}

```

Il modello di risposta per `getObject` restituisce i risultati:

```
$util.toJson($ctx.result)
```

Ecco come `listObjects` potrebbe apparire un modello di richiesta. A `listObjects` utilizza un'operazione `Scan` per restituire uno o più elementi e attributi. Tieni presente che questo frammento di codice non ha una tabella Amazon DynamoDB configurata come origine dati. Questo viene usato solo come esempio:

```

{
  "version" : "2017-02-28",
  "operation" : "Scan",
}

```

Il modello di risposta per `listObjects` restituisce i risultati:

```
$util.toJson($ctx.result.items)
```

Di seguito sono riportati alcuni esempi di utilizzo di questo schema con le query GraphQL:

```

mutation CreateObject {
  putObject(email: "example@example.com"
    json: "{\"a\":1, \"b\":3, \"string\": 234}"
    date: "1970-01-01Z"
    time: "12:00:34."
    datetime: "1930-01-01T16:00:00-07:00"
    timestamp: -123123
    url:"https://amazon.com"
    phoneno: "+1 555 764 4377"
    ip: "127.0.0.1/8"
  ) {
    id
    email
  }
}

```

```
        json
        date
        time
        datetime
        url
        timestamp
        phoneno
        ip
    }
}

query getObject {
  getObject(id:"0d97daf0-48e6-4ffc-8d48-0537e8a843d2"){
    email
    url
    timestamp
    phoneno
    ip
  }
}

query listObjects {
  listObjects {
    json
    date
    time
    datetime
  }
}
```

Interfacce e unioni in GraphQL

[Il sistema di tipo GraphQL supporta le interfacce.](#) Un'interfaccia espone un determinato set di campi che deve essere incluso in un tipo per implementare l'interfaccia.

[Il sistema di tipo GraphQL supporta anche Unions.](#) Le unioni sono identiche alle interfacce, ad eccezione del fatto che non definiscono un set comune di campi. Le unioni sono in genere preferibili rispetto alle interfacce quando i tipi possibili non condividono una gerarchia logica.

La sezione seguente è un riferimento per la digitazione dello schema.

Esempi di interfacce

Potremmo rappresentare un'Eventinterfaccia che rappresenta qualsiasi tipo di attività o incontro di persone. Alcuni tipi di eventi possibili sono `ConcertConference`, e `Festival`. Questi tipi condividono tutti caratteristiche comuni, in quanto hanno tutti un nome, un luogo in cui avviene l'evento e una data di inizio e una di fine. Anche questi tipi presentano delle differenze; a `Conference` offre un elenco di relatori e workshop, mentre a `Concert` presenta una band che si esibisce.

In Schema Definition Language (SDL), l'Eventinterfaccia è definita come segue:

```
interface Event {  
    id: ID!  
    name : String!  
    startsAt: String  
    endsAt: String  
    venue: Venue  
    minAgeRestriction: Int  
}
```

E ciascuno dei tipi implementa l'Eventinterfaccia come segue:

```
type Concert implements Event {  
    id: ID!  
    name: String!  
    startsAt: String  
    endsAt: String  
    venue: Venue  
    minAgeRestriction: Int  
    performingBand: String  
}  
  
type Festival implements Event {  
    id: ID!  
    name: String!  
    startsAt: String  
    endsAt: String  
    venue: Venue  
    minAgeRestriction: Int  
    performers: [String]  
}
```

```
type Conference implements Event {  
  id: ID!  
  name: String!  
  startsAt: String  
  endsAt: String  
  venue: Venue  
  minAgeRestriction: Int  
  speakers: [String]  
  workshops: [String]  
}
```

Le interfacce sono utili per rappresentare elementi che possono essere di diversi tipi. Ad esempio, possiamo cercare tutti gli eventi che avvengono in un luogo specifico. Aggiungiamo un campo `findEventsByVenue` allo schema come segue:

```
schema {  
  query: Query  
}  
  
type Query {  
  # Retrieve Events at a specific Venue  
  findEventsAtVenue(venueId: ID!): [Event]  
}  
  
type Venue {  
  id: ID!  
  name: String  
  address: String  
  maxOccupancy: Int  
}  
  
type Concert implements Event {  
  id: ID!  
  name: String!  
  startsAt: String  
  endsAt: String  
  venue: Venue  
  minAgeRestriction: Int  
  performingBand: String  
}  
  
interface Event {  
  id: ID!
```

```
    name: String!
    startsAt: String
    endsAt: String
    venue: Venue
    minAgeRestriction: Int
  }

type Festival implements Event {
  id: ID!
  name: String!
  startsAt: String
  endsAt: String
  venue: Venue
  minAgeRestriction: Int
  performers: [String]
}

type Conference implements Event {
  id: ID!
  name: String!
  startsAt: String
  endsAt: String
  venue: Venue
  minAgeRestriction: Int
  speakers: [String]
  workshops: [String]
}
```

`findEventsByVenue` restituisce un elenco di `Event`. Poiché i campi dell'interfaccia GraphQL sono comuni a tutti i tipi di implementazione, è possibile selezionare qualsiasi campo nell'interfaccia `Event` (`id`, `name`, `startsAt`, `endsAt`, `venue` e `minAgeRestriction`). Inoltre, puoi accedere ai campi in qualsiasi tipo di implementazione utilizzando [frammenti](#) GraphQL, purché specifichi il tipo.

Esaminiamo un esempio di query GraphQL che utilizza l'interfaccia.

```
query {
  findEventsAtVenue(venueId: "Madison Square Garden") {
    id
    name
    minAgeRestriction
    startsAt

    ... on Festival {
```

```
    performers
  }

  ... on Concert {
    performingBand
  }

  ... on Conference {
    speakers
    workshops
  }
}
```

La query precedente produce un singolo elenco di risultati e il server può, per impostazione predefinita, ordinare gli eventi in base alla data di inizio.

```
{
  "data": {
    "findEventsAtVenue": [
      {
        "id": "Festival-2",
        "name": "Festival 2",
        "minAgeRestriction": 21,
        "startsAt": "2018-10-05T14:48:00.000Z",
        "performers": [
          "The Singers",
          "The Screammers"
        ]
      },
      {
        "id": "Concert-3",
        "name": "Concert 3",
        "minAgeRestriction": 18,
        "startsAt": "2018-10-07T14:48:00.000Z",
        "performingBand": "The Jumpers"
      },
      {
        "id": "Conference-4",
        "name": "Conference 4",
        "minAgeRestriction": null,
        "startsAt": "2018-10-09T14:48:00.000Z",
        "speakers": [
```

```

    "The Storytellers"
  ],
  "workshops": [
    "Writing",
    "Reading"
  ]
}
]
}
}
}

```

Poiché i risultati vengono restituiti come una singola raccolta di eventi, l'utilizzo di interfacce per rappresentare caratteristiche comuni è molto utile per l'ordinamento dei risultati.

Esempi di unione

Come affermato in precedenza, i sindacati non definiscono insiemi di campi comuni. Un risultato di ricerca potrebbe rappresentare molti tipi diversi. Utilizzando lo schema Event, è possibile definire un'unione SearchResult come segue:

```

type Query {
  # Retrieve Events at a specific Venue
  findEventsAtVenue(venueId: ID!): [Event]
  # Search across all content
  search(query: String!): [SearchResult]
}

union SearchResult = Conference | Festival | Concert | Venue

```

In questo caso, per interrogare qualsiasi campo sulla nostra SearchResult unione, devi usare i frammenti:

```

query {
  search(query: "Madison") {
    ... on Venue {
      id
      name
      address
    }

    ... on Festival {
      id

```



```
    name
    performers
  }

  ... on Concert {
    id
    name
    performingBand
  }

  ... on Conference {
    speakers
    workshops
  }
}
```

Digita la risoluzione in AWS AppSync

La risoluzione dei tipi è il meccanismo attraverso il quale il motore GraphQL identifica un valore risolto come tipo di oggetto specifico.

Tornando all'esempio della ricerca sindacale, a condizione che la nostra query abbia prodotto risultati, ogni elemento nell'elenco dei risultati deve presentarsi come uno dei possibili tipi definiti dall'`SearchResult` unione (ovvero,, `Conference FestivalConcert`, o `Venue`).

Poiché la logica per identificare un tipo `Festival` rispetto a un tipo `Venue` o `Conference` dipende dai requisiti dell'applicazione, il motore GraphQL deve ricevere un suggerimento per identificare i possibili tipi dai risultati non elaborati.

Con AWS AppSync, questo suggerimento è rappresentato da un metacampo denominato `__typename`, il cui valore corrisponde al nome del tipo di oggetto identificato. `__typename` è obbligatorio per i tipi restituiti che sono interfacce o unioni.

Esempio di risoluzione dei tipi

Ora, riutilizzeremo lo schema precedente. Puoi continuare passando alla console e aggiungendo il codice seguente nella pagina Schema:

```
schema {
  query: Query
}
```

```
type Query {  
  # Retrieve Events at a specific Venue  
  findEventsAtVenue(venueId: ID!): [Event]  
  # Search across all content  
  search(query: String!): [SearchResult]  
}  
  
union SearchResult = Conference | Festival | Concert | Venue  
  
type Venue {  
  id: ID!  
  name: String!  
  address: String  
  maxOccupancy: Int  
}  
  
interface Event {  
  id: ID!  
  name: String!  
  startsAt: String  
  endsAt: String  
  venue: Venue  
  minAgeRestriction: Int  
}  
  
type Festival implements Event {  
  id: ID!  
  name: String!  
  startsAt: String  
  endsAt: String  
  venue: Venue  
  minAgeRestriction: Int  
  performers: [String]  
}  
  
type Conference implements Event {  
  id: ID!  
  name: String!  
  startsAt: String  
  endsAt: String  
  venue: Venue  
  minAgeRestriction: Int  
  speakers: [String]
```

```
workshops: [String]
}

type Concert implements Event {
  id: ID!
  name: String!
  startsAt: String
  endsAt: String
  venue: Venue
  minAgeRestriction: Int
  performingBand: String
}
```

Collegiamo un resolver al campo `Query.search`. Nella `Resolvers` sezione, scegliete `Allega`, create una nuova origine dati di tipo `NONE`, quindi assegnatele un nome `StubDataSource`. Ai fini di questo esempio, fingeremo di aver recuperato i risultati da un'origine esterna e di aver codificato tali risultati nel modello di mappatura della richiesta.

Nel riquadro del modello di mappatura della richiesta immetti i seguenti dati:

```
{
  "version" : "2018-05-29",
  "payload":
  ## We are effectively mocking our search results for this example
  [
    {
      "id": "Venue-1",
      "name": "Venue 1",
      "address": "2121 7th Ave, Seattle, WA 98121",
      "maxOccupancy": 1000
    },
    {
      "id": "Festival-2",
      "name": "Festival 2",
      "performers": ["The Singers", "The Screammers"]
    },
    {
      "id": "Concert-3",
      "name": "Concert 3",
      "performingBand": "The Jumpers"
    },
    {
      "id": "Conference-4",
```

```

        "name": "Conference 4",
        "speakers": ["The Storytellers"],
        "workshops": ["Writing", "Reading"]
      }
    ]
  }
}

```

Se l'applicazione restituisce il nome del tipo come parte del id campo, la logica di risoluzione dei tipi deve analizzare il id campo per estrarre il nome del tipo e quindi aggiungere il __typename campo a ciascuno dei risultati. È possibile eseguire la logica nel modello di mappatura della risposta come segue:

Note

Puoi eseguire questa attività anche come parte della tua funzione Lambda, se utilizzi l'origine dati Lambda.

```

#foreach ($result in $context.result)
  ## Extract type name from the id field.
  #set( $typeName = $result.id.split("-")[0] )
  #set( $ignore = $result.put("__typename", $typeName))
#end
$util.toJson($context.result)

```

Eseguire la seguente query:

```

query {
  search(query: "Madison") {
    ... on Venue {
      id
      name
      address
    }

    ... on Festival {
      id
      name
      performers
    }
  }
}

```

```
... on Concert {
  id
  name
  performingBand
}

... on Conference {
  speakers
  workshops
}
}
```

La query restituisce i seguenti risultati:

```
{
  "data": {
    "search": [
      {
        "id": "Venue-1",
        "name": "Venue 1",
        "address": "2121 7th Ave, Seattle, WA 98121"
      },
      {
        "id": "Festival-2",
        "name": "Festival 2",
        "performers": [
          "The Singers",
          "The Screammers"
        ]
      },
      {
        "id": "Concert-3",
        "name": "Concert 3",
        "performingBand": "The Jumpers"
      },
      {
        "speakers": [
          "The Storytellers"
        ],
        "workshops": [
          "Writing",
          "Reading"
        ]
      }
    ]
  }
}
```

```
    ]  
  }  
]  
}  
}
```

La logica di risoluzione dei tipi varia a seconda dell'applicazione. Ad esempio, puoi definire una logica di identificazione diversa che controlla l'esistenza di determinati campi o addirittura una combinazione di campi. Ovvero, puoi rilevare la presenza del campo `performers` per identificare un tipo `Festival` o la combinazione dei campi `speakers` e `workshops` per identificare un tipo `Conference`. In definitiva, sta a te definire la logica che desideri utilizzare.

Risoluzione dei problemi ed errori comuni in AWS AppSync

Questa sezione illustra alcuni errori comuni e come risolvere i problemi che li riguardano.

Mappatura della chiave di DynamoDB errata

Se l'operazione GraphQL restituisce il seguente messaggio di errore, è possibile che la struttura del modello di mappatura della richiesta non corrisponda alla struttura delle chiavi di Amazon DynamoDB:

```
The provided key element does not match the schema (Service: AmazonDynamoDBv2; Status Code: 400; Error Code
```

Ad esempio, se la tabella DynamoDB ha una chiave hash "id" chiamata e il modello "PostID" dice, come nell'esempio seguente, ciò genera l'errore precedente, perché non corrisponde. "id" "PostID"

```
{
  "version" : "2017-02-28",
  "operation" : "GetItem",
  "key" : {
    "PostID" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
  }
}
```

Resolver mancante

Se si esegue un'operazione di GraphQL, ad esempio una query, e si ottiene una risposta nulla, questo potrebbe dipendere dal fatto che non si dispone di un resolver configurato.

Ad esempio, se si importa uno schema che definisce un campo `getCustomer(userId: ID!)`: e non è stato configurato un resolver per questo campo, quando si esegue una query, ad esempio `getCustomer(userId:"ID123"){...}`, si riceverà una risposta simile alla seguente:

```
{
  "data": {
    "getCustomer": null
  }
}
```

```
}  
}
```

Errori del modello di mappatura

Se il modello di mappatura non è configurato correttamente, si riceverà una risposta di GraphQL il cui `errorType` è `MappingTemplate`. Il campo `message` deve indicare dov'è il problema nel modello di mappatura.

Ad esempio, se non si dispone di un campo `operation` nel modello di mappatura della richiesta, oppure se il nome del campo `operation` non è corretto, si otterrà una risposta simile alla seguente:

```
{  
  "data": {  
    "searchPosts": null  
  },  
  "errors": [  
    {  
      "path": [  
        "searchPosts"  
      ],  
      "errorType": "MappingTemplate",  
      "locations": [  
        {  
          "line": 2,  
          "column": 3  
        }  
      ],  
      "message": "Value for field '$[operation]' not found."  
    }  
  ]  
}
```

Tipi restituiti non corretti

Il tipo restituito dall'origine dati deve corrispondere al tipo definito di un oggetto nello schema. In caso contrario, è possibile che venga visualizzato un errore di GraphQL come ad esempio:

```
"errors": [  
  {
```



```
    "path": [
      "posts"
    ],
    "locations": null,
    "message": "Can't resolve value (/posts) : type mismatch error, expected type LIST,
got OBJECT"
  }
]
```

Ad esempio questo potrebbe verificarsi con la seguente definizione di query:

```
type Query {
  posts: [Post]
}
```

La quale prevede un ELENCO di oggetti `[Posts]`. Ad esempio, se si disponesse di una funzione Lambda nel Node.JS del tipo simile al seguente:

```
const result = { data: data.Items.map(item => { return item ; }) };
callback(err, result);
```

Questo genererebbe un errore poiché `result` è un oggetto. È necessario modificare il callback in `result.data` oppure modificare lo schema per non restituire un ELENCO.

Elaborazione di richieste non valide

Se non AWS AppSync è in grado di elaborare e inviare una richiesta (a causa di dati impropri, ad esempio una sintassi non valida) al field resolver, il payload di risposta restituirà i dati del campo con i valori impostati su e gli eventuali errori pertinenti. `null`

Le traduzioni sono generate tramite traduzione automatica. In caso di conflitto tra il contenuto di una traduzione e la versione originale in Inglese, quest'ultima prevarrà.