Developer Guide

AWS AppSync Events

AWS AppSync Events: Developer Guide

Table of Contents

What is AWS AppSync Events?

AWS AppSync Events lets you create secure and performant serverless WebSocket APIs that can broadcast real-time event data to millions of subscribers, without you having to manage connections or resource scaling. With AWS AppSync Events, there is no API code required to get started, so you can create production-ready real-time web and mobile experiences in minutes.

AWS AppSync Events further simplifies the management and scaling of real-time applications by shifting tasks like message transformation and broadcast, publish and subscribe authentication, and the creation of logs and metrics to AWS, while delivering reduced time to market, low latency, enhanced security, and lower total costs.

With Event APIs, you can enable the following network communication types.

- Unicast
- Multicast
- Broadcast
- Batch publishing and messaging

This allows you to build the following types of interactive and collaborative experiences.

- Live chat and messaging
- Sports and score updates
- Real-time in-app alerts and notifications
- Live commenting and activity feeds

AWS AppSync Events simplifies real-time application development by providing the following features.

- Automatic management of WebSocket connections and scaling
- Built-in support for broadcasting events to large numbers of subscribers
- Flexible event filtering and transformation capabilities
- Fine-grained authentication and authorization
- Seamless integration with other AWS services, data sources, and external systems for event-driven architectures

Whether you're building a small prototype or a large-scale production application, AWS AppSync Events enables you to incorporate real-time experiences using a fully managed and scalable platform, so you can focus on your application logic instead of the undifferentiated heavy lifting of managing infrastructure.

AWS AppSync Events features

WebSocket and HTTP Support

Clients can publish events over HTTP or WebSocket, and can subscribe to channels using WebSockets.

Event APIs provide WebSocket endpoints that enable real-time and pub/sub capabilities.

Channel namespaces and channels

Events are published to channels, that are grouped under namespaces.

Namespaces allow you to define authentication and authorization rules and serverless functions that apply to all channels within that namespace.

Namespace handlers

You can use the following two types of handlers to configure functions that run in response to publish and subscribe actions.

- OnPublish - Runs when an event is published to a channel, allowing you to transform, filter, and reject events.
- OnSubscribe - Runs when a client subscribes to a channel, allowing you to customize the behavior or reject the subscription request.

Flexible authentication and authorization

Event APIs supports various authentication mechanisms (API key, IAM, Amazon Cognito, OIDC, and Lambda authorizers) that can be configured at the API level and customized at the channel namespace level.

Channel subscriptions

Clients receive events for channels they are subscribed to.

Wildcard Channel Subscriptions

Clients can subscribe to a group of related channels using a wildcard syntax (e.g., "namespace/channel/*"), allowing them to receive events from multiple channels without explicitly subscribing to each one.

Scalable Event Broadcasting

The Event API automatically scales to handle large numbers of concurrent connections and can efficiently broadcast events to all subscribed clients.

Integration with the AWS Ecosystem

AWS AppSync Events integrates with other AWS services like Amazon CloudWatch Logs, CloudWatch metrics, and AWS WAF. You can easily implement event-driven architectures by publishing directly from services like Amazon EventBridge, and AWS Lambda. Amazon Cognito is directly supported as an authorization type. AWS AppSync Event APIs can be configured to interact with multiple AWS data sources, enabling you to process and route events efficiently.

Pricing for AWS AppSync Events

When you use AWS AppSync Events, you pay only for what you use with no minimum fees or mandatory service usage. For more information, see AWS AppSync pricing.

AWS AppSync Events concepts

Before you get started, review the following topics to help you understand the fundamental concepts of AWS AppSync Events.

API

An Event API provides real-time capabilities by enabling you to publish events over HTTP and WebSocket, and subscribe to events over WebSockets. An Event API has one or more channel namespaces that define the capabilities and behavior of channels that events can be addressed to. To learn more about configuring an API, see Configuring authorization and authentication to secure Event APIs and Configuring custom domain names for Event APIs.

Event

An event is a JSON-formatted unit of data that can be published to channels on your API and received by clients that are interested in that channel. Events can contain any arbitrary data you want to transmit in real-time, such as user actions, data updates, system notifications, or sensor readings. Events are designed to be lightweight and efficient, with a maximum size of 240 KB per event.

Channel

Channels are the routing mechanism for directing events from publishers to subscribers. You can think of a channel as a "topic" or "subject" that represents a stream of related events. Clients subscribe to channels in order to receive events published to those channels in real-time. Channels are ephemeral and can be created on-demand.

Channel namespace

A channel namespace (or just namespace for short) provides a way to define the capabilities and behaviors of the channels associated with it. Each namespace has a name. This name represents the starting segment (the prefix) of a channel path. Any channel where the first segment in the path matches the name of a namespace, belongs to that namespace. For example, any channel

with a first segment of `default`, such as `/default/messages`, `/default/greetings`, and `/default/inbox/user` belongs to the namespace named `default`. To learn more about namespaces, see Understanding channel namespaces.

Event handler

An event handler is a function defined in a namespace. An event handler is a custom function to process published events before they are broadcast to subscribers. A event handler can also be used to process subscription requests when clients try to subscribe to a channel. Handlers are written in JavaScript and run on the AppSync_JS runtime. Handlers can be associated with data sources to access external data or run custom business logic.

Data sources

Data sources are the resources that AWS AppSync Events can interact with to process, store, or retrieve event data. AWS AppSync Events supports data sources that provide the following capabilities:

- **Storage solutions (DynamoDB, Amazon RDS)** – Store event data and state.
- **Compute resources (Lambda)** – Process and transform events.
- **AI/ML services (Amazon Bedrock)** – Add intelligence to event processing.
- **Search and analytics (OpenSearch)** – Index and analyze event patterns.
- **Event routing (EventBridge)** – Connect with broader event architectures.
- **External integration (HTTP endpoints)** – Interact with external services.

Data sources can be associated with channel namespaces as integrations, and used by handlers to access external data. AWS AppSync Events can combine multiple data sources within a single API, enabling you to build sophisticated event processing workflows. Each event can interact with one or more data sources based on your application's needs.

Publishing

Publishing is the act of sending a batch of events to your Event API. Published events can be broadcasted to subscribed clients. Publish is done over HTTP or WebSocket.

Subscribing

Subscribing is the act of listening for events on a specific channel or subset of channels over an Event API WebSocket. Clients that subscribe can receive broadcast events in real-time. Clients can establish multiple subscriptions over a single WebSocket.

Getting started with AWS AppSync Events

You can quickly get started with AWS AppSync Events by creating an AWS AppSync Event API and accessing it from a client. The first tutorial in this section will guide you through creating your first AWS AppSync Event API in the AWS AppSync console. Then you will learn to publish and subscribe to your event. The second tutorial guides you through creating a React app with Vite and an Amplify client. Then you will use this Amplify client to publish and subscribe messages.

Topics

Prerequisites

Before you begin the getting started tutorials, confirm that you have completed the prerequisites to get set up with an AWS account.

Sign up for an AWS account

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

1. Open https://portal.aws.amazon.com/billing/signup.
2. Follow the online instructions.

   Part of the sign-up procedure involves receiving a phone call or text message and entering a verification code on the phone keypad.

   When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign administrative access to a user, and use only the root user to perform tasks that require root user access.

AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to https://aws.amazon.com/ and choosing **My Account**.

Create a user with administrative access

After you sign up for an AWS account, secure your AWS account root user, enable AWS IAM Identity Center, and create an administrative user so that you don't use the root user for everyday tasks.

Secure your AWS account root user

1. Sign in to the AWS Management Console as the account owner by choosing **Root user** and entering your AWS account email address. On the next page, enter your password.

   For help signing in by using root user, see Signing in as the root user in the *AWS Sign-In User Guide*.

2. Turn on multi-factor authentication (MFA) for your root user.

   For instructions, see Enable a virtual MFA device for your AWS account root user (console) in the *IAM User Guide*.

Create a user with administrative access

1. Enable IAM Identity Center.

   For instructions, see Enabling AWS IAM Identity Center in the *AWS IAM Identity Center User Guide*.

2. In IAM Identity Center, grant administrative access to a user.

   For a tutorial about using the IAM Identity Center directory as your identity source, see Configure user access with the default IAM Identity Center directory in the *AWS IAM Identity Center User Guide*.

Sign in as the user with administrative access

- To sign in with your IAM Identity Center user, use the sign-in URL that was sent to your email address when you created the IAM Identity Center user.

For help signing in using an IAM Identity Center user, see Signing in to the AWS access portal in the *AWS Sign-In User Guide*.

Assign access to additional users

1. In IAM Identity Center, create a permission set that follows the best practice of applying least-privilege permissions.

   For instructions, see Create a permission set in the *AWS IAM Identity Center User Guide*.
2. Assign users to a group, and then assign single sign-on access to the group.

   For instructions, see Add groups in the *AWS IAM Identity Center User Guide*.

Account credentials

Although you can use your root user credentials to access AWS AppSync, we recommend that you use an AWS Identity and Access Management (IAM) account instead. You can use the AWSAppSyncAdministrator managed policy to grant your IAM account the correct permissions to manage your AWS AppSync resources.

Set up the AWS Command Line Interface

You can use the AWS Command Line Interface (CLI) to manage your AWS AppSync resources. For information about how to install and configure the AWS CLI, see Getting started with the AWS CLI in the *AWS Command Line Interface User Guide*.

Creating an AWS AppSync Event API

AWS AppSync Events allows you to create Event APIs to enable real-time capabilities in your applications. In this section, you create an API with a default channel namespace. You'll then use the AWS AppSync console to publish messages and subscribe to messages sent to channels in the namespace.

In this tutorial you will complete the following tasks.

Topics

Step 1: Create an event API using the AWS AppSync console

1. Sign in to the AWS Management Console and open the AWS AppSync console.
2. On the AWS AppSync console service page, choose **Create API**, then choose **Event API**.
3. On the **Create Event API** page, in the **API details** section, do the following:
   a. For **API** enter the name of your API.
   b. (optional) Enter the contact name for the API.
4. Choose **Create**.

You have now created an Event API. The API is configured with API Key as an authorization mode for connect, publish, and subscribe actions. A default channel namespace with the name “default” has also been created.

To learn more about customizing authorization, see Configuring authorization and authentication to secure Event APIs. To learn more about channel namespaces, see Understanding channel namespaces

Step 2: Publish and subscribe to receive your first event

Use the following instructions to publish an event.

1. In the AWS AppSync console choose the **Pub/Sub Editor** tab for the Event API that you created in step 1.
2. In the **Publish** section, for **Channel** enter **`default`** in the first text box and enter **`/messages`** in the second text box.
3. In the code editor, enter the following JSON payload.

```
[{"message": "Hello world!"}]
```

4. Choose **Publish**.
5. The publisher logs table displays a response similar to the following to confirm success.

```
{
  "failed": [],
  "successful": [
    {
      "identifier": "53287bee-ae0d-42e7-8a90-e9d2a49e4bd7",
      "index": 0
    }
  ]
}
```

Now use the following instructions to subscribe to recieve messages.

1. In the **Subscribe** section of the editor, choose **Connect** to connect to your WebSocket endpoint.
2. For **Channel**, enter the name of the channel you want to subscribe to, **`/default/messages`**, and then choose **Subscribe**.
3. In the code editor, choose **Publish** again. You should receive a new data event in the subscriber logs table with the published event.

Step 3: Use wildcards in your channel subscription

You can specify a wildcard "*" at the end of a channel path to receive events published to all channels that match. Use the following instructions to set up a wildcard channel subscription for your Event API.

1. If you are still subscribed to the channel from step 2, choose **Unsubscribe**.
2. In the **Subscribe** section, for **Channel** enter **`/default/*`**.
3. In the code editor section, choose **Publish** again to send another event. You should receive a new data event in the subscriber logs table
4. In the **Publish** section, change the **Channel** name to **`/default/greetings/tutorial`**.
5. Choose **Publish**. You receive the message in the **Subscribe** section

Step 4: Publish in batches

You can publish events in batches of up to five. Subscribed clients receive each message individually.

1. In the AWS AppSync console, continue in the **Pub/Sub Editor** tab using the Event API that you were working with in step 3.
2. In the **Publish** section JSON code editor, enter the following:

```
[
  {"message": "Hello world!"},
  {"message": "Bonjour lemonde!"},
  "Hola Mundo!"
]
```

3. Choose **Publish**.
4. In the **Subscribe** log table, you receive 3 data events.

Getting started with the Amplify Events client

You can connect to your AWS AppSync Event API using any HTTP and WebSocket client, and you can also use the Amplify client for JavaScript. This getting started tutorial guides you through connecting to an Event API from a JavaScript React application.

In this tutorial you will complete the following tasks.

Topics

Step 1: Create an event API

1. Sign in to the AWS Management Console and open the AWS AppSync console.
2. On the AWS AppSync console service page, choose **Create API**, then choose **Event API**.

3. On the **Create Event API** page, in the **API details** section, do the following:

   a. For **API** enter the name of your API.

   b. (optional) Enter the contact name for the API.

4. Choose **Create**.

You have now created an Event API. The API is configured with API Key as an authorization mode for connect, publish, and subscribe actions. A default channel namespace with the name “default” has also been created.

To learn more about customizing authorization, see Configuring authorization and authentication to secure Event APIs. To learn more about channel namespaces, see Understanding channel namespaces

Step 2: Deploy a React app with Vite

1. From your local work environment, run the following command in a terminal window to create a new Vite app for React.

```
npm create vite@latest appsync-events-app -- --template react
```

2. Run the following command to switch to the `appsync-events-app` directory and install the dependencies and the Amplify library.

```
cd appsync-events-app
npm install
npm install aws-amplify
```

3. Open a different terminal window and `cd` into your `appsync-events-app` directory. Run the following command to start your sever in dev mode.

```
npm run dev
```

(Optional) Configure Tailwind CSS

You can set up Tailwind CSS to style your project.

1. Open a terminal window and run the following commands to install the dependencies.

```
npm install -D tailwindcss postcss autoprefixer
npx tailwindcss init -p
```

2. Update the `tailwind.config.js` file with the following code.

```
/** @type {import('tailwindcss').Config} */
export default {
  content: [
    "./index.html",
    "./src/**/*.{js,ts,jsx,tsx}",
  ],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

3. Set the content of `./src/index.css` to the following.

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

4. Use the following command to start and restart your Vite app from the terminal where it is currently running.

```
npm run dev
```

Step 3: Configure the Amplify client

1. Sign in to the AWS Management Console and open the AWS AppSync console.
2. Open the **Integration** tab for the Event API that you created in step 1.
3. Download your configuration file.
4. Save the `amplify_outputs.json` file in your project's `src` directory. Your configuration file will look like the following.

```
{
  "API": {
```

```
    "Events": {
      "endpoint": "https://abc1234567890.aws-appsync.us-west-2.amazonaws.com/
event",
      "region": "us-west-2",
      "defaultAuthMode": "apiKey",
      "apiKey": "da2-your-api-key-1234567890"
    }
  }
}
```

> **Important**
>
> You must set `defaultAuthMode` to `apiKey` and *not* `API_KEY`.

Step 4: Connect to a channel and receive messages

1. Update your `App.jsx` file with the following code.

```
import { useEffect, useState, useRef } from 'react'
import './App.css'

import { Amplify } from 'aws-amplify'
import { events } from 'aws-amplify/data'
import config from './amplify_outputs.json'

Amplify.configure(config)

export default function App() {
  const [messages, setMessages] = useState([])
  const [room, setRoom] = useState('')
  const counterRef = useRef(null)

  useEffect(() => {
    if (!room || !room.length) {
      return
    }
    let timeoutID
    const pr = events.connect(`/default/${room}`)
    pr.then((channel) => {
      channel.subscribe({
        next: (data) => {
```

```
          setMessages((messages) => [...messages, data.message])
          if (timeoutID) {
            clearTimeout(timeoutID);
          }
          counterRef.current?.classList.add('animate-bounce')
          timeoutID = setTimeout(() => {
            counterRef.current?.classList.remove('animate-bounce')
          }, 3000);
        },
        error: (value) => console.error(value),
      })
    })

    return () => {
      pr?.then((channel) => channel?.close())
    }
  }, [room])

  return (
    <div className='max-w-screen-md mx-auto'>
      <h2 className='my-4 p-4 font-semibold text-xl'>AppSync Events - Messages</h2>
      <button
        type="button"
        className='border rounded-md px-4 py-2 items-center text-sm font-medium
 transition-colors focus-visible:outline-none focus-visible:ring-1 focus-
visible:ring-ring disabled:pointer-events-none disabled:opacity-50 bg-sky-200
 shadow hover:bg-sky-200/90'
        onClick={() => {
          const room = prompt('Room:')
          if (room && room.length) {
            setMessages([])
            setRoom(room.trim().replace(/\W+/g, '-'))
          }
        }}
      >
        set room
      </button>
      <div className='my-4 border-b-2  border-sky-500 py-1 flex justify-between'>
        <div>
          {room ? (
            <span>
              Currently in room: <b>{room}</b>
            </span>
          ) : (
```

```
            <span>Pick a room to get started</span>
          )}
        </div>
        <div className='flex items-center uppercase text-xs tracking-wider font-
semibold'>
          <div className='mr-2'>Messages count:</div>
          <span ref={counterRef} className='transition-all inline-flex
 items-center rounded-md bg-sky-100 px-2.5 py-0.5 text-xs font-medium text-
sky-900'>{messages.length}</span></div>
      </div>
      <section id="messages" className='space-y-2'>
        {messages.map((message, index) => (
          <div
            key={index}
            className='border-b py-1 flex justify-between px-2'
          ><div>
              {message}
            </div>
            <div> </div>
          </div>
        ))}
      </section>
    </div>
  )
}
```

2. Open the app in your browser and choose the **set room** button to set the room to "greetings".
3. Open the AWS AppSync console, and go to the **Pub/Sub Editor** tab for your API.
4. Set your channel to `/default/greetings`. Choose the "default" namespace and set the rest of the path to `/greetings`.
5. Paste the following into the editor to send the event.

```
[
    {
        "message": "hello world!"
    }
]
```

6. You should see the message in your app.
7. Choose the **set room** button again to select another room. Send another event in the Pub/Sub Editor to the `/default/greetings` channel. You do not see that message in your app.

Step 5: Send a message from your app

1. Open your App component and add the following line of code at the top of the file.

```
const [message, setMessage] = useState('')
```

2. In the same file, locate the last `section` element and add the following code.

```
<section>
        <form
          disabled={!room}
          className='w-full flex justify-between mt-8'
          onSubmit={async (e) => {
            e.preventDefault()
            const event = { message }
            setMessage('')
            await events.post(`/default/${room}`, event)
          }}
        >
          <input
            type="text"
            name="message"
            placeholder="Message:"
            className='flex flex-1 rounded-md border border-input px-3
 py-1 h-9 text-sm shadow-sm transition-colors focus-visible:outline-none
 focus-visible:ring-1 focus-visible:ring-ring disabled:cursor-not-allowed
 disabled:opacity-50 bg-transparent'
            value={message}
            disabled={!room}
            onChange={(e) => setMessage(e.target.value)}
          />
          <button
            type="submit"
            className='ml-4 border rounded-md px-4 flex items-center text-sm font-
medium transition-colors focus-visible:outline-none focus-visible:ring-1 focus-
visible:ring-ring disabled:pointer-events-none disabled:opacity-50 bg-sky-200
 shadow hover:bg-sky-200/90'
            disabled={!room || !message || !message.length}>
            <svg xmlns="http://www.w3.org/2000/svg" className='size-4' width="24"
 height="24" viewBox="0 0 24 24" fill="none" stroke="currentColor" strokeWidth="2"
 strokeLinecap="round" strokeLinejoin="round"><path d="M14.536 21.686a.5.5 0 0
 0 .937-.024l6.5-19a.496.496 0 0 0-.635-.635l-19 6.5a.5.5 0 0 0-.024.937l7.93
 3.18a2 2 0 0 1 1.112 1.11z" /><path d="m21.854 2.147-10.94 10.939" /></svg>
```

```
          </button>
        </form>
      </section>
```

3. You can now send a message to a room that you select, directly from your app.

AWS AppSync Event API tutorials

To help you understand how AWS AppSync Events works, the following tutorials walk you through common tasks and workflows.

Tutorials

- Persist user messages with a DynamoDB table integration using AppSyncJs
- Create a wscat clone using NodeJs and IAM auth using AppSyncJs
- Publishing from a NodeJS Lambda function with IAM auth using AppSyncJs

Persist user messages with a DynamoDB table integration using AppSyncJs

In this tutorial, you'll learn how to create an AWS AppSync Events API that stores user messages in a DynamoDB table as they are published and before they are broadcasted to connected users. You'll create a simple messaging system where users can send messages and then later retrieve their message history.

You will complete this tutorial using only the AWS Management Console. Before you begin, complete the following set up prerequisites.

Sign up for an AWS account

If you are not already an AWS customer, you need to create an AWS account by following the online instructions. Signing up enables you to access AWS AppSync and other AWS services that you can use with your AWS AppSync GraphQL and Event APIs.

Understand how to access the AWS Management Console

The AWS AppSync console is available at https://console.aws.amazon.com/appsync.

Understand the AWS AppSync and DynamoDB services

For more information about DynamoDB, see What is Amazon DynamoDB in the *Amazon DynamoDB Developer Guide*.

Step 1: Create a DynamoDB-backed AWS AppSync GraphQL API

In this step, you create an AWS AppSync GraphQL API along with the DynamoDB table to store the messages.

Create a GraphQL API

1. Sign in to the AWS Management Console and open the AWS AppSync console at AWS AppSync console.
2. On the AWS AppSync home page, choose **Create API**, then choose **GraphQL API**.
3. On the **Select API type** page, choose **Design from scratch**. Then choose **Next**.
4. In the **Specify API details** section, for **API name**, enter **`graphql-messages-api`**. Then choose **Next**.
5. On the **Specify GraphQL resources** page, in the **Create a GraphQL type** section, choose **Create type backed by a DynamoDB table now**.
6. In the **Configure model information** section, do the following:
   a. For **Model name**, enter **`Message`**.
   b. For **Fields**, do the following to add a channel field to your model:
      i. Choose **Add new field**.
      ii. For **Name**, enter **`channel`**.
      iii. For **Type**, choose **String**.
      iv. For **Array**, choose **No**.
      v. For **Required**, choose **Yes**.
   c. Do the following to add an id field to your model:
      i. Choose **Add new field**.
      ii. For **Name**, enter **`id`**.
      iii. For **Type**, choose **String**.
      iv. For **Array**, choose **No**.
      v. For **Required**, choose **Yes**.
   d. Do the following to add a user field to your model:
      i. Choose **Add new field**.

ii. For **Name**, enter **`user`**.

iii. For **Type**, choose **String**.

iv. For **Array**, choose **No**.

v. For **Required**, choose **Yes**.

e. Do the following to add a content field to your model:

i. Choose **Add new field**.

ii. For **Name**, enter **`content`**.

iii. For **Type**, choose **String**.

iv. For **Array**, choose **No**.

v. For **Required**, choose **Yes**.

f. Do the following to add a createdAt field to your model:

i. Choose **Add new field**.

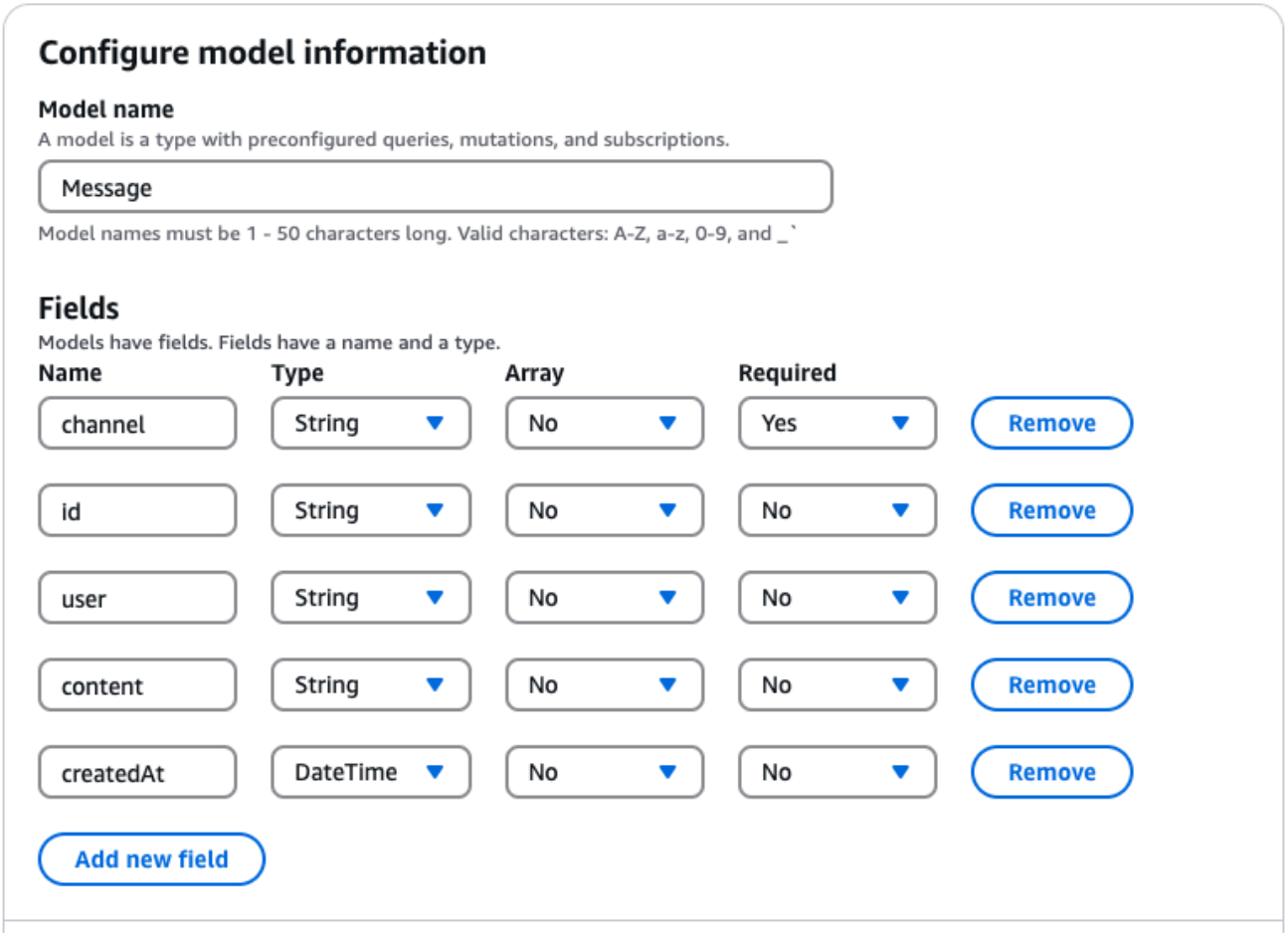
ii. For **Name**, enter **`createdAt`**.

iii. For **Type**, choose **DateTime**.

iv. For **Array**, choose **No**.

v. For **Required**, choose **Yes**.

The following screenshot shows how your fields should be configured:



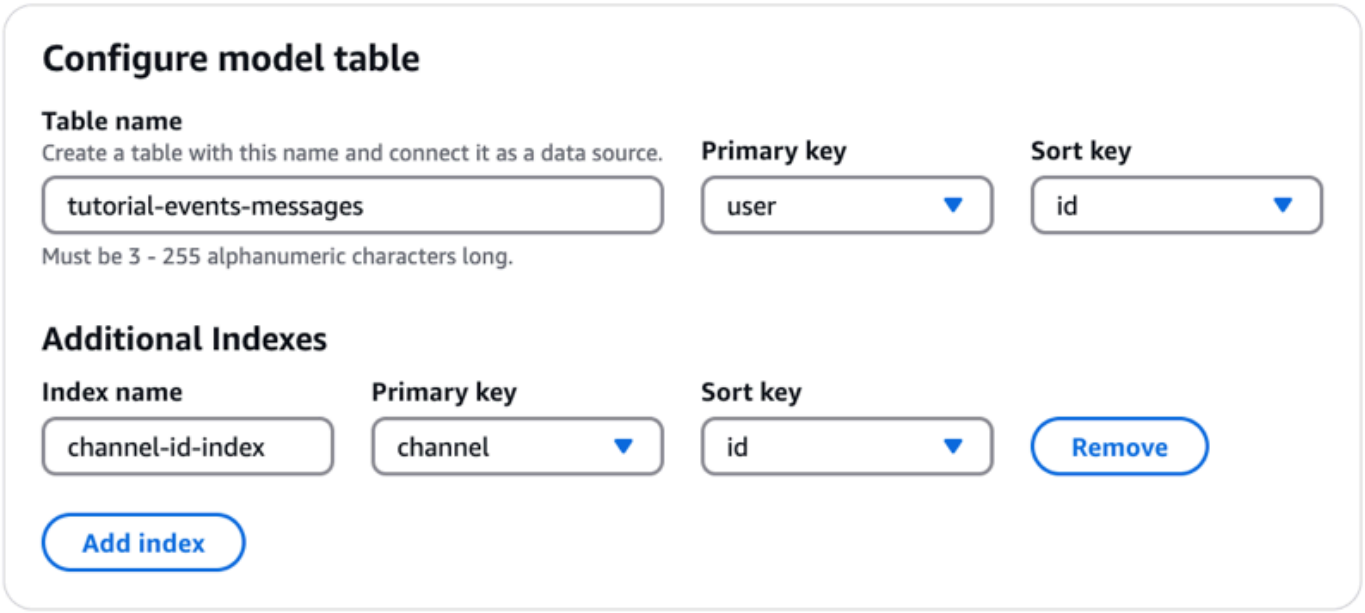
7. **In the Configure model table** section, do the following:

   a. For **Table name** enter **`tutorial-events-messages`**.

   b. For **Primary key**, select **user**.

   c. For **Sort key**, select **id**.

   The following screenshot shows how your fields should be configured:



8. Choose **Next**.
9. On the **Review and create** page, review your API details, and choose **Create API**.

   The AWS AppSync console creates a DynamoDB table and an AWS AppSync GraphQL API to query the table. Wait for the API creation process to complete before continuing to Step 2 of this tutorial.

Step 2: Create an AWS AppSync Events API

In this step, you create the AWS AppSync Events API that you can use to publish messages. Other clients will be able to subscribe to a channel on the API to receive messages.

Create the AWS AppSync Events API

1. In the left navigation menu, choose **API** to return to the **APIs** page.
2. In the **API types** section, choose **Create Event API**.
3. On the **Create Event API** page, in the **API details** section, for **API name**, enter **`events-messages-api`**.
4. Choose **Create**.

   The AWS AppSync console creates an Event API with API key authorization mode configured, and a channel namespace called `default`.

Next, create a data source for your DynamoDB table that you created in Step 1.

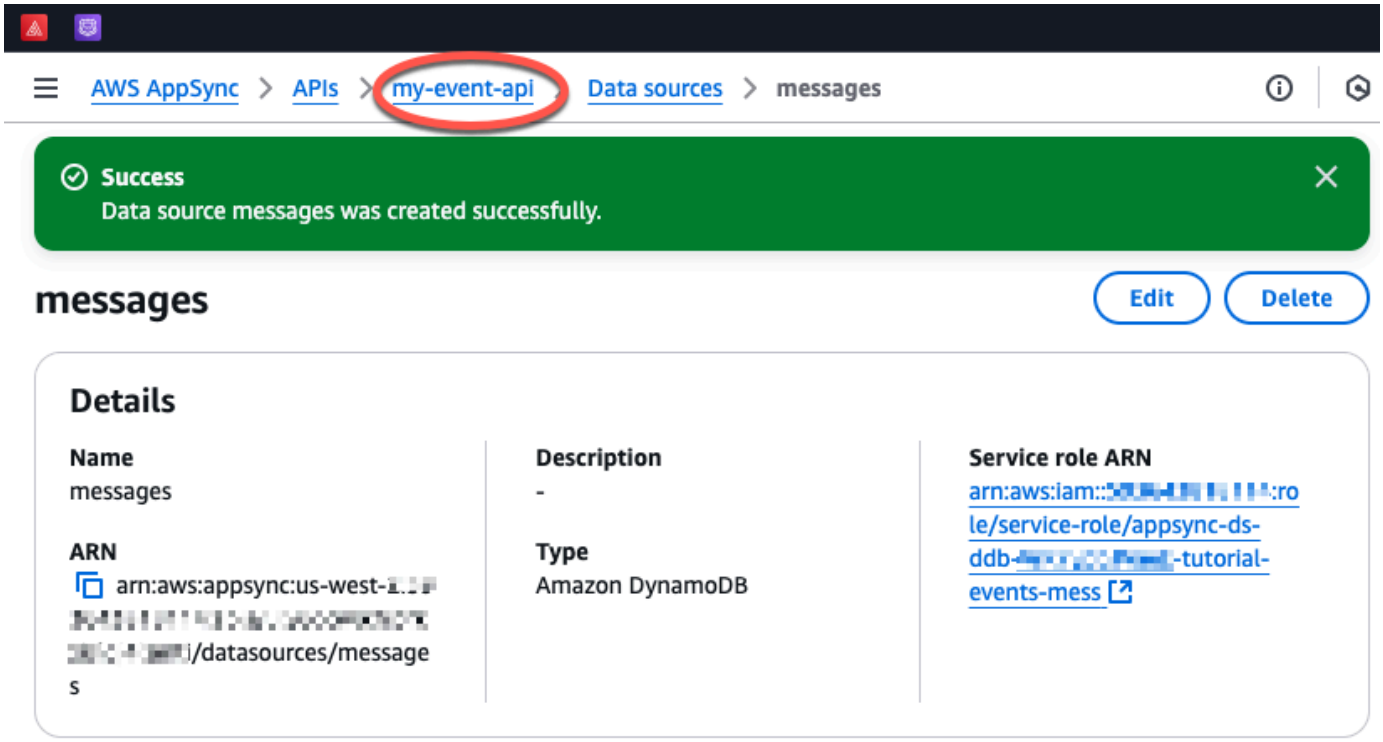
To create a data source for the DynamoDB table

1. Choose the **Data sources** tab.
2. Choose **Create data source**.
3. On the **Create data source** page, do the following:
   a. For **Name**, enter **`messages`**.
   b. For **Data source type**, choose **Amazon DynamoDB**.
   c. The configuration automatically selects the current AWS Region.
   d. For **Table name**, select **`tutorial-events-messages`** from the list.
4. Choose **Create**.

Now you can create your data source integration in your namespace.

To create a data source integration with a namespace

1. From your **messages** data source page, you need to return to the settings page for **my-event-api**. You can either use your browser's back button or you can choose **my-event-api** from the top menu in the console window. The following screenshot shows the location of the menu with **my-event-api** circled.



2. Choose the **Namespaces** tab, then choose the **default** namespace.
3. Choose **Edit**.
4. On the **Edit default** page, in the **Handler** section, choose **Code with data source**.
5. For **Publish configuration**, **Data source name**, choose **messages**. For **Behavior**, choose **Code**.
6. For **Subscribe configuration**, leave the **Data source name** blank.
7. In the code editor, enter the following code:

```
import { util } from '@aws-appsync/utils'
import * as ddb from '@aws-appsync/utils/dynamodb'

export const onPublish = {
  request(ctx) {
    const channel = ctx.info.channel.path
    const createdAt = util.time.nowISO8601()
    return ddb.batchPut({
      tables: {
        'tutorial-events-messages': ctx.events.map(({ payload }) => {
          return { channel, id: util.autoKsuid(), createdAt, ...payload }
        }),
      },
    })
  },
```

```
  // simply forward the events for broadcast
  response: (ctx) => ctx.events
}
```

The code uses the ddb.batchPut function to save multiple events to the "tutorial-events-messages" using a DynamoDB BatchWrite operation. It adds the channel path, creates an id, and a timestamp before saving the items.

8. Choose **Save** to confirm your code changes.

Step 3: Test the AWS AppSync Events API

To test the API

1. From your **default** namespace page, you need to return to the settings page for **my-event-api**. You can either use your browser's back button or you can choose **my-event-api** from the top menu in the console window.
2. On the **my-event-api** page, choose the **Pub/Sub Editor** tab.
3. In the **Subscrib** section, choose **Connect** and **Subscribe** to the default/* channel.
4. In the **Publish** section, paste the following in the code editor:

```
{
  "user": "john",
  "content": "Hello, world!"
}
```

5. Choose **Publish**.
6. Update the content in the editor with the following:

```
[
  {
    "user": "sarah",
    "content": "Working on a fresh batch of cookies!"
  },
  {
    "user": "harry",
    "content": "Yum, can't wait!"
  }
]
```

7. Choose **Publish** again. After each time you publish, the events are displayed in the **Subscribe** table.

Step 4: Retrieve your messages

You can view your messages directly in the DynamoDB table. However, there is a simple process for querying the data with GraphQL. This step demonstrates how to do this using your **graphql-messages-api** AWS AppSync GraphQL API.

To query data with a GraphQL API

1. From your **my-event-api Pub/Sub Editor** page, you need to go to your **graphql-messages-api**. In the left navigation, choose **APIs**. In the **APIs** section, choose **graphql-messages-api**.
2. In the left navigation menu, choose **Queries**.
3. On the **Queries** page, clear the contents of the editor and enter the following query.

```
query byChannel {
  queryMessagesByChannelIdIndex(channel: "/default/channel", first: 10) {
    items {
      channel
      id
      user
      content
      createdAt
    }
  }
}
```

4. Choose the orange **Execute query** button on the upper right. The query returns the first ten items in your DynamoDB table.

Step 5: (Optional) Delete the resources you created

If you no longer need the resources that you created for this tutorial, you can delete them. This step helps ensure that you aren't charged for resources that you aren't using.

You can delete all of the following resources directly in the AWS AppSync console:

- The AWS AppSync Events API **my-event-api**

- The AWS AppSync GraphQL API **graphql-messages-api**
- The DynamoDB table **tutorial-events-messages**

Create a wscat clone using NodeJs and IAM auth using AppSyncJs

This tutorial shows you how to create a wscat clone that enables real-time messaging using AWS AppSync Events with IAM authorization using Smithy libraries with TypeScript in NodeJS.

Before you begin

Make sure you have completed the prerequisites in the Getting Started topic. You will also need to use the AWS CLI.

Your profile must have permissions to run the following actions. For more information about AWS AppSync actions, see Actions, resources, and condition keys for AWSAWS AppSync.

- `appsync:EventConnect` - Grants permission to connect to an Event API
- `appsync:EventPublish` - Grants permission to publish events to a channel namespace
- `appsync:EventSubscribe` - Grants permission to subscribe to a channel namespace

You will also need to have *NodeJS* working in your environment. You need NodeJS version 22.14.0 or higher.

> **Note**
>
> You can download the latest version of NodeJs here or use a tool like Node Version Manager (nvm).

The tutorial

Implementation steps

1. **Create an Event API**

   To begin, create the Event API you will interact with. For more information, see Creating an AWS AppSync Event API to create the *Event API*.

2. **Configure authorization**

   Once created, sign into the AWS AppSync console and configure the IAM authorization in **Settings**.

   a. In the **Authorization modes** section, choose **Add**. On the next screen, choose **AWS Identity and Access Management (IAM)**, and choose **Add** to use IAM as an authorization mode.

   b. In the **Authorization configuration** section, choose **Edit**.

   c. On the next page, add **IAM authorization** to the **Connection authorization modes**, the **default publish authorization** modes, and the **default subscribe authorization modes**. Choose **Update** to save your changes.

3. **Start a new NodeJs project**

   In your terminal, create a new directory and initialize project:

   ```
   mkdir eventscat
   cd eventscat
   npm init -y
   ```

4. **Install TypeScript packages**

   Install the required TypeScript packages and bundle them using esbuild

   ```
   npm install esbuild typescript
   npm install -D @tsconfig/node20 @types/node
   ```

5. **Create a new signer library for IAM**

   When using IAM as an authorization mode, you must sign your request using Sigv4. Create a *signer library* handles that task. Start by installing the required packages.

```
npm i @aws-crypto/sha256-js \
  @aws-sdk/credential-providers \
  @smithy/protocol-http \
  @smithy/signature-v4
```

Create a new file: `src/signer.ts` with the following code

```
import { Sha256 } from '@aws-crypto/sha256-js'
import { fromNodeProviderChain } from '@aws-sdk/credential-providers'
import { HttpRequest } from '@smithy/protocol-http'
import { SignatureV4 } from '@smithy/signature-v4'

/** AppSync Events WebSocket sub-protocol identifier */
export const AWS_APPSYNC_EVENTS_SUBPROTOCOL = 'aws-appsync-event-ws'

/** Default headers required for AppSync Events API requests */
export const DEFAULT_HEADERS = {
  accept: 'application/json, text/javascript',
  'content-encoding': 'amz-1.0',
  'content-type': 'application/json; charset=UTF-8',
}

/**
 * Prepares signed material for a request
 * @param httpDomain - the Events API HTTP domain
 * @param region - the API region
 * @param body - the body to sign
 * @returns signed material for a request
 */
export async function sign(httpDomain: string, region: string, body: string) {
  const credentials = fromNodeProviderChain()

  const signer = new SignatureV4({
    credentials,
    service: 'appsync',
    region,
    sha256: Sha256,
  })

  const url = new URL(`https://${httpDomain}/event`)
  const httpRequest = new HttpRequest({
    method: 'POST',
```

```
    headers: { ...DEFAULT_HEADERS, host: url.hostname },
    body,
    hostname: url.hostname,
    path: url.pathname,
  })

  const signedReq = await signer.sign(httpRequest)
  return { host: signedReq.hostname, ...signedReq.headers }
}

/**
 * Get the HTTP domain and region or null if it cannot be identified
 * @param wsDomain - the websocket domain
 */
function getHttpDomain(wsDomain: string) {
  const pattern =
    /^\w{26}\.appsync-realtime-api.(\w{2}(?:(?:-\w{2,})+)-\d)\.amazonaws.com(?:
\.cn)?$/

  const match = wsDomain.match(pattern)
  if (match) {
    return {
      httpDomain: wsDomain.replace('appsync-realtime-api', 'appsync-api'),
      region: match[1],
    }
  }
  return null
}

/**
 * Transforms an object into a valid based64Url string
 * @param auth - header material
 */
function getAuthProtocol(auth: unknown): string {
  const based64UrlHeader = btoa(JSON.stringify(auth))
    .replace(/\+/g, '-') // Convert '+' to '-'
    .replace(/\//g, '_') // Convert '/' to '_'
    .replace(/=+$/, '') // Remove padding `=`
  return `header-${based64UrlHeader}`
}

/**
 * Gets the protocol array that authorizes connecting to an API
 * @param wsDomain -the WebSocket endpoint domain
```

```
 * @param region - the AWS region of the API
 * @returns
 */
export async function getAuthForConnect(wsDomain: string, region?: string) {
  const domain = getHttpDomain(wsDomain)
  if (!domain && !region) {
    throw new Error('Must provide region when using a custom domain')
  }
  const httpDomain = domain?.httpDomain ?? wsDomain
  const _region = domain?.region ?? region!
  const signed = await sign(httpDomain, _region, '{}')
  const protocol = getAuthProtocol(signed)
  return [AWS_APPSYNC_EVENTS_SUBPROTOCOL, protocol]
}

/**
 * Gets the authorization header for a websocket message
 * @param wsDomain -the WebSocket endpoint domain
 * @param body -the request to sign
 * @param region - the AWS region of the API
 * @returns
 */
export async function getAuthForMessage(wsDomain: string, body: unknown, region?:
 string) {
  const domain = getHttpDomain(wsDomain)
  if (!domain && !region) {
    throw new Error('Must provide region when using a custom domain')
  }
  const httpDomain = domain?.httpDomain ?? wsDomain
  const _region = domain?.region ?? region!
  return await sign(httpDomain, _region, JSON.stringify(body))
}
```

The signer uses `fromNodeProviderChain` to retrieve your local credentials and uses SigV4 to sign the request headers and content.

6. **Create your program**

   a. We recommend to you use the chalk library to provide some color on your terminal and the commander library to define and parse your program command options.

   ```
   npm install chalk commander
   ```

b. Create the file `src/eventscat.ts` with the following code for your program.

```
import chalk from 'chalk'
import { program } from 'commander'
import EventEmitter from 'node:events'
import readline from 'node:readline'
import { getAuthForConnect, getAuthForMessage } from './signer'

/**
 * InputReader - processes console input.
 *
 * @extends EventEmitter
 */
class Console extends EventEmitter {
  stdin: NodeJS.ReadStream & { fd: 0 }
  stdout: NodeJS.WriteStream & { fd: 1 }
  stderr: NodeJS.WriteStream & { fd: 2 }
  readlineInterface: readline.Interface

  constructor() {
    super()

    this.stdin = process.stdin
    this.stdout = process.stdout
    this.stderr = process.stderr

    this.readlineInterface = readline.createInterface({
      input: process.stdin,
      output: process.stdout,
    })

    this.readlineInterface
      .on('line', (data) => {
        this.emit('line', data)
      })
      .on('close', () => {
        this.emit('close')
      })
  }

  prompt() {
    this.readlineInterface.prompt(true)
  }
```

```
  message(msg: string) {
    const payload = JSON.parse(msg)
    this.clear()
    if (payload.type === 'ka') {
      this.stdout.write(chalk.magenta('<ka>\n'))
    } else if (payload.type === 'data') {
      this.stdout.write(chalk.blue(`<
 ${JSON.stringify(JSON.parse(payload.event), null, 2)}\n`))
    } else {
      this.stdout.write(chalk.bgBlack.white(`* ${payload.type} *\n`))
    }
    this.prompt()
  }

  print(msg: any) {
    this.clear()
    this.stdout.write(msg + '\n')
    this.prompt()
  }

  clear() {
    this.stdout.write('\r\u001b[2K\u001b[3D')
  }
}

export async function main() {
  program
    .version('demo')
    .option('-r, --realtime <domain>', 'AppSync Events real-time domain')
    .option('-s, --subscribe <channel>', 'Channel to subscribe to')
    .option('-p, --publish [channel]', 'Channel to publish to')
    .parse(process.argv)

  const options = program.opts()
  if (!options.realtime || !options.subscribe) {
    return program.help()
  }

  const wsConsole = new Console()
  const domain = options.realtime
  const channel = options.subscribe

  const protocols = await getAuthForConnect(domain)
```

```
  const ws = new WebSocket(`wss://${domain}/event/realtime`, protocols)

  ws.onopen = async () => {
    ws.send(
      JSON.stringify({
        type: 'subscribe',
        id: crypto.randomUUID(),
        channel,
        authorization: await getAuthForMessage(domain, { channel }),
      }),
    )
    wsConsole.print(chalk.green('Connected (press CTRL+C to quit)'))
    wsConsole.on('line', async (data: string) => {
      if (!data || !data.trim().length || !options.publish) {
        return wsConsole.prompt()
      }
      const channel = options.publish
      const events = [JSON.stringify(data.trim())]
      ws.send(
        JSON.stringify({
          type: 'publish',
          id: crypto.randomUUID(),
          channel,
          events,
          authorization: await getAuthForMessage(domain, { channel, events }),
        }),
      )
      wsConsole.prompt()
    })
  }

  ws.onclose = (event) => {
    wsConsole.print(chalk.green(`Disconnected (code: ${event.code}, reason:
"${event.reason}")`))
    wsConsole.clear()
    process.exit()
  }

  ws.onerror = (err) => {
    wsConsole.print(chalk.red(err))
    process.exit(-1)
  }

  ws.onmessage = (data) => wsConsole.message(data.data)
```

```
  wsConsole.on('close', () => {
    ws.close()
    process.exit()
  })
}
```

c. Specify the scripts property in `package.json` file.

```
...
    "scripts": {
    "build": "esbuild --platform=node --target=node20 --bundle --outfile=bin/
eventscat.js src/eventscat.ts"
  },
...
```

d. Build the code.

```
npm run build
```

e. Add the file `bin/eventscat`.

```
#!/usr/bin/env node

const { main } = require('./eventscat.js')
main()
```

Change the mode of the file to execute it

```
chmod +x bin/eventscat
```

7. **Using the program**

Test the implementation by connecting to your API. To do this, subscribe to `/default/*` *channel*, and publish on the `/default/iam` *channel*. You can find the realtime domain of your API in the **Settings** section of the AWS AppSync console under **Realtime**.

```
./bin/eventscat --realtime 1234567890.appsync-api.us-east-2.amazonaws.com \
--subscribe "/default/*" --publish "/default/iam"
```

Once the client is connected, you can start publishing messages simply by entering text and pressing **enter**. You also start receiving messages published to your **subscribe** channel.

8. **(Optional) Clean up**

   Once you are done with this tutorial, you can delete the API you created by going to the AWS AppSync console, selecting the API and choosing **Delete**.

Publishing from a NodeJS Lambda function with IAM auth using AppSyncJs

AWS AppSync Events allows you to create Event APIs to enable real-time capabilities in your applications. In this tutorial, you use AWS CDK to create your API and a Lambda function that publishes messages. You'll use IAM auth as the authorization mode to publish to your configured channel.

Before you begin

To get started, make sure you have gone through the *prerequisites*. You will need an AWS account. You will also work from the command line.

Installing AWS CDK and creating up your project

The AWS Cloud Development Kit (AWS CDK) (AWS CDK) is an open-source software development framework for defining cloud infrastructure as code with modern programming languages and deploying it through AWS CloudFormation.

From your terminal, install CDK.

```
npm install -g aws-cdk
```

Next, create a new folder for your application

> **Note**
>
> Be sure to use the exact name as AWS CDK uses the folder name to name your app.

```
mkdir publish-from-lambda
```

From the `publish-from-lambda` directory, init a new app:

```
cd publish-from-lambda
cdk init app --language typescript
```

The AWS CDK CLI creates a AWS CDK app containing a single AWS CDK stack. You'll be using TypeScript in this project, so install `esbuild` to bundle your code:

```
npm i -D esbuild@0
```

Now, update the `lib/publish-from-lambda-stack.ts` file with this code:

```
import * as cdk from 'aws-cdk-lib'
import * as appsync from 'aws-cdk-lib/aws-appsync'
import { Runtime } from 'aws-cdk-lib/aws-lambda'
import * as nodejs from 'aws-cdk-lib/aws-lambda-nodejs'
import type { Construct } from 'constructs'
import * as path from 'node:path'

export class PublishFromLambdaStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props)

    const { IAM, API_KEY } = appsync.AppSyncAuthorizationType
    const apiKeyProvider = { authorizationType: API_KEY }
    const iamProvider = { authorizationType: IAM }

    // 1. Create a new AppSync Events API
    const api = new appsync.EventApi(this, 'api', {
      apiName: 'messages-api',
      authorizationConfig: {
        authProviders: [apiKeyProvider, iamProvider],
        connectionAuthModeTypes: [API_KEY],
        defaultSubscribeAuthModeTypes: [API_KEY],
        defaultPublishAuthModeTypes: [IAM],
      },
    })

    // 2. Create a channel namespace called `messages`
```

```
    const ns = api.addChannelNamespace('messages')

    // 3. Create the Lambda function: publisher
    const publisher = new nodejs.NodejsFunction(this, 'publisher', {
      entry: path.join(__dirname, 'lambda', 'publisher.ts'),
      runtime: Runtime.NODEJS_22_X,
      timeout: cdk.Duration.minutes(1),
      bundling: { externalModules: ['@aws-sdk/*'] },
      environment: {
        NAMESPACE: 'messages',
        HTTP_ENDPOINT: api.httpDns,
      },
    })

    // 4. Grant publisher IAM permissions to publish messages to the namespace
    ns.grantPublish(publisher)

    // 5. Output the API ID and the function name
    new cdk.CfnOutput(this, 'apiId', { value: api.apiId })
    new cdk.CfnOutput(this, 'fnName', { value: publisher.functionName })
  }
}
```

Let's recap the stack implementation:

- You create a new AWS AppSync Events API called `messages-api`. With this API, you can connect to the WebSocket endpoint and subscribe to a channel using an API Key. You can only publish to a channel using IAM authorization.
- You create a name space called `messages`. This allows you to publish and subscribes to channel paths like starting with `/messages`.
- You create a NodeJS Lambda function running on `NODEJS_22`. The function can run up to 1 minute. You configure the environment variable and pass the API's http endpoint and namespace.
- You grant the function permissions to publish to any channel in the namespace.
- Finally, you output the id of the API and the name of the function. This makes it easy to reference them later, and find the resources in the console.

Create your Lambda function

Next, create you'll create a Lambda function that can publish events to the provided channel using IAM authorization. To do so, you will use the *ob-appsync-events-request* library. This small library implements a `Request` object that is signed with the available IAM credentials. You can review the implementation on Github.

In your `publish-from-lambda` directory, create a new file `publisher.ts` in the `lib/lambda` folder and install the library.

```
mkdir -p lib/lambda
touch lib/lambda/publisher.ts
npm i -D ob-appsync-events-request @types/aws-lambda
```

Update `publisher.ts` with the following code.

```
import type { Handler } from 'aws-lambda'
import { PublishRequest } from 'ob-appsync-events-request'

const ENV = process.env as {
  NAMESPACE: string
  HTTP_ENDPOINT: string
}

export const handler: Handler = async (event) => {
  const createdAt = new Date().toISOString()

  // Create a signed request
  const request = await PublishRequest.signed(
    `https://${ENV.HTTP_ENDPOINT}/event`,
    `${ENV.NAMESPACE}/lambda`,
    { message: 'Hello, world!', createdAt },
    { message: 'Bonjour le monde!', createdAt },
    { message: '¡Hola Mundo!', createdAt },
  )

  // Send the request using fetch
  const response = await fetch(request)
  const result = await response.json()
  console.log(result)
  return result
}
```

When triggered, the Lambda function publishes 3 messages to the channel path `/messages/lambda`. The request is signed using the function's IAM permissions. This is possible because you granted the permissions to the function in your AWS CDK stack configuration!

Deploy the stack

You can deploy the stack at this point from your `publish-from-lambda` folder:

```
npm run cdk deploy
```

Review the prompts to proceed with the deployment, once done, you should see the output:

```
Outputs:
PublishFromLambdaStack.apiId = your_api_id
PublishFromLambdaStack.fnName = your_function_name
```

Subscribe and publish

Open the AWS AppSync console in the region you deployed your stack in. On the APIs page, you can locate your API by pasting in your `apiId` value in the search bar under *APIs*. Select your messages-api, then select *Pub/Sub Editor*. Scroll down to the *Subscribe* section, and choose *Connect*. Under *Channel*, you choose `messages`, then choose *Subscribe*.

Back in your terminal, invoke the Lambda function with the AWS CLI, using the output `fnName`.

```
aws lambda invoke \
  --function-name "your_function_name" \
  --output text /dev/stdout
```

The function handler is executed, and in your AWS AppSync Events console, you receive the 3 messages!

Cleaning up

Once you are done with this tutorial, you can clean up your resources by running the command:

```
npm run cdk destroy
```

Understanding channel namespaces

Channel namespaces (or just namespaces for short) define the channels that are available on your Event API, and the capabilities and behaviors of these channels. Channel namespaces provide a scalable approach to managing large numbers of channels. Instead of configuring each channel individually, developers can apply settings across an entire namespace.

Each namespace has a name. This name represents the starting segment (the prefix) of a channel path. Any channel where the first segment in the path matches the name of a namespace, belongs to that namespace. For example, any channel with a first segment of `default`, such as `/default/messages`, `/default/greetings`, and `/default/inbox/user` belongs to the namespace named `default`. A channel namespace is made up of a maximum of five segments.

In a sports-related Event API example, you could have namespaces such as `basketball`, `soccer`, and `baseball`, with channels within each namespace such as `basketball/games/1`, `soccer/scores`, and `baseball/players`.

You can only publish and subscribe to channels that belong to a defined namespace. However, a channel is ephemeral and is created on-demand when a client needs to publish or subscribe to it.

When you define your Event API, you must configure the default authorization for connecting, publishing and subscribing to an Event API. The publishing and subscribing authorization configuration is automatically applied to all namespaces. You can override this configuration at the namespace. To learn more about authorization, see Configuring authorization and authentication to secure Event APIs.

Process real-time events with AWS AppSync event handlers

Topics

Overview

AWS AppSync Event handlers let you run custom business logic on real-time events.

These JavaScript functions:

- Run on the AWS AppSync runtime
- Process published events
- Authorize subscription requests

This topic covers *simple* event handlers without data source integration. For information about integrating with data sources like DynamoDB tables and Lambda functions, see the following topics:

- Data source integrations
- Direct Lambda integrations

> **Note**
>
> Event handlers run after the incoming request is authorized through your configured authorization mode.

onPublish handler

Use the `onPublish` handler to process and filter events before they reach subscribers. This handler runs each time an event is published to a channel.

The handler uses this signature:

```
function onPublish(context: Context): OutgoingEvent[] | null
```

```
type Context = {
  events: IncomingEvent[];  // Array of events to process
  channel: string;          // Channel the events were published to
  identity: Identity;       // Information about the publisher
  info: {
    channel: {
      path: string;        // Full channel path
      segments: string[];  // Path segments
    }
  }
```

```
  channelNamespace: {
    name: string
  }
  operation: 'SUBSCRIBE' | 'PUBLISH'
}
```

The handler receives a context object with:

- `events` - Array of events to process
- `channel` - Target channel name
- `identity` - Publisher information

For more information on the context object reference, see the Context Reference guide.

Common onPublish tasks

Forward all events

The default API behavior for the `onPublish` handler *forwards all received events*. The onPublish function needs a context object as a parameter.

```
export function onPublish(ctx) {
  return ctx.events
}
```

Filter specific events

Filter out events and return only those matching specific criteria. In the following example, the handler filters the events and only forwards those that have *odds greater than 0.5*.

```
export function onPublish(ctx) {
  return ctx.events.filter((event) => event.payload.odds > 0.5)
}
```

Transform events

Transform events by mapping them to a new shape. In the following example, the handler below formats each event to include a *timestamp* and changes the message to upper case format.

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

```
export function onPublish(ctx) {
  return ctx.events.map(event => ({
    id: event.id,
    payload: {
      ...event.payload,
      message: event.payload.message.toUpperCase(),
      timestamp: util.time.nowISO8601()
    }
  }))
}
```

Important rules for onPublish

- Avoid duplicate event IDs
- Events with an `error` property won't broadcast
- `Null` values in the returned array are ignored
- Returns an array of events or null
- Match each returned event ID to an incoming event
- Using an unknown event ID will result in an error

onSubscribe handler

Use the onSubscribe handler to authorize and process subscription requests before allowing channel subscriptions. The handler in the following example runs when the client subscribes.

The handler uses this signature:

```
function onSubscribe(context: Context): void
```

For more information on the context object reference, see the Context Reference Guide.

Examples

Example Logging a subscription request

In the following example, a Amazon Cognito user pool authorization is used and the onSubscribe handler logs a message when an admin user subscribes to a channel.

```
export function onSubscribe(ctx) {
```

```
  if (ctx.identity.groups.includes('admin')) {
    console.log(`Admin ${ctx.identity.sub} subscribed to ${ctx.channel}`)
  }
}
```

Example Reject a request

In the following example, a subscription request is rejected by calling `util.unauthorized()`. The onSubscribe handler restricts the Amazon Cognito user pool's authenticated users to their own channel. Clients can only subscribe to channels that match the pattern `/messages/inbox/[username]`.

```
export function onSubscribe(ctx) {
  const requested = ctx.info.channel.path
  const allowed = `/messages/inbox/${ctx.identity.username}`

  if (requested !== allowed) {
    util.unauthorized()
  }
}
```

Error handling

Use error handling to manage invalid requests and provide meaningful feedback to users. This toipic explains how to implement error handling for both *HTTP endpoints* and *WebSocket connections*. To reject requests and return errors to publishers or subscribers, use the `utility.error` function. When the publishing is done using the HTTP endpoint, it returns an HTTP `403` response.

> **Note**
>
> - When publishing is done over an HTTP endpoint, it returns an HTTP `403` response.
> - When publishing over WebSocket, it returns a `publish_error` message with the provided message.

Examples

The following examples demonstrates how to return an error message.

Example Validating required message properties

```
export function onPublish(ctx) {
  return ctx.events.map(event => {
    if (!event.payload.message) {
      event.error = "Message required"
    }
    return event
  })
}
```

Example Rejecting entire requests

```
export function onPublish(ctx) {
  util.error("Operation not allowed")
}
```

Data source integrations for AWS AppSync events

With AWS AppSync Events, you can create event handlers that run custom business logic on AWS AppSync's JavaScript runtime. By configuring data source integrations, you can interact with external systems like Lambdafunctions, DynamoDB tables, or Amazon RDS Aurora databases.

Topics

Overview

You can integrate data sources in your channel namespaces to interact with DynamoDB tables and Lambda functions. Each integration requires implementing request and response functions. This topic explains how to configure and use data sources with event handlers.

For detailed information about data source configuration options, see Working with data sources for AWS AppSync Event APIs.

Handler structure

A handler that interacts with a data source has the following two functions:

- Request — Defines the payload sent to the data source
- Response — Defines the payload sent to the data source

Example Saving events to DynamoDB before broadcasting

The following example defines an `onPublish` handler that saves all published events to a messages_table in DynamoDB before forwarding them to be broadcast.

```
import * as ddb from `@aws-appsync/utils/dynamodb`

const TABLE = 'messages_table'
export const onPublish = {
  request(ctx) {
    const channel = ctx.info.channel.path
    return ddb.batchPut({
      tables: {
        [TABLE]: ctx.events.map(({ id, payload }) => ({ channel, id, ...payload })),
      },
    })
  },
  response(ctx) {
    console.log(`Batch Put result:`, ctx.result.data[TABLE])
    return ctx.events
  }
}
```

> **Note**
>
> You must return the list of events you want broadcasted in the response function.

Example Forward all events after accessing it from the data source

```
import * as ddb from `@aws-appsync/utils/dynamodb`

const TABLE = 'messages_table'
```

```
export const onPublish = {
  request(ctx) {
    const channel = ctx.info.channel.path
    return ddb.batchPut({
      tables: {
        [TABLE]: ctx.events.map(({ id, payload }) => ({ channel, id, ...payload })),
      },
    })
  },
  response: (ctx) => ctx.events // forward all events
}
```

onSubscribe handler

Use the onSubscribe handler to process subscription requests.

Example Logging subscriptions to an RDS database

```
import { insert, createPgStatement as pg } from '@aws-appsync/utils/rds'

export const onSubscribe = {
  request(ctx) {
    const values = {
      channel: ctx.info.channel.path,
      identity: ctx.identity
    }
    return pg(insert({table: 'subs', values}))
  },
  response(ctx) {} // Required empty function
}
```

> **Note**
>
> You must provide a response function, which can be empty if no other action is required.

Example Granting access based on a channel path

```
import { select, createPgtatement as pg, toJsonObject } from '@aws-appsync/utils/rds';

export const onSubscribe = {
```

```
  request(ctx) {
    const values = {
      channel: ctx.info.channel.path,
      identity: ctx.identity
    }
    return pg(insert({
      select: 'subs',
      where: { channel: { eq: 'ALLOWED' }}
    }))
  },
  response(ctx){
    const { error, result } = ctx;
    if (error) {
        return util.error('db error')
    }
    const res = toJsonObject(result)[1][0];
    if (!res.length) {
      // did not find the item, subscription is not authorized
      util.unauthorized()
    }
  }
}
```

Example Skipping the data source

To skip the data source invocation at runtime use the `runtime.earlyReturn` utility. The `earlyReturn` operation returns the provided payload and skips the response function.

```
import * as ddb from `@aws-appsync/utils/dynamodb`

export const onPublish = {
  request(ctx) {
    if (ctx.info.channel.segments.includes('private')) {
      // return early and do no execute the response.
      return runtime.earlyReturn(ctx.events)
    }
    const channel = ctx.info.channel.path
    const event = ctx.events[0]
    const key = { channel, id: event.payload.id }
    return ddb.put({ key, item: event.payload })
  },
  response: (ctx) => ctx.events // forward all events
}
```

Direct data source integrations using Lambda

AWS AppSync lets you integrate Lambda functions directly with your channel namespace without writing additional handler code. Both *publish* and *subscribe* operations are supported through `Request/Response` mode and `event` mode.

> **Note**
>
> Configuring your direct Lambda integration in `event` mode triggers your Lambda aysncronously and does not wait for a reply. The result of the invocation does not impact the rest of the `onPublish` or `onSubscribe` handling.

How it works

Your Lambda function receives a context object containing event information. The function then passes a context object containing information for the events. The Lambda function can perform the following operations.

- Filter and transform events for broadcasting
- Return error messages for failed processing
- Handle both publish and subscribe operations

Example Setting a Lambda configuration of a channel namespace in AWS CloudFormation

Set up a direct Lambda integration, configure `handlerConfigs` on your channel namespace using AWS CloudFormation.

```
{
  "Type" : "AWS::AppSync::ChannelNamespace",
  "Properties" : {
    "ApiId" : "api-id-123",
    "Name" : "lambda-direct-ns",
    "HandlerConfigs" : {
      "OnPublish": {
        "Behavior": "DIRECT",
        "Integration": {
          "DataSourceName": "LAMBDA_FUNCTION_DS",
          "LambdaConfig": {
```

```
            "InvokeType": "REQUEST_RESPONSE"
          }
        }
      }
    }
  }
}
```

Example Setting a Lambda configuration of a channel namespace in the AWS Cloud Development Kit (AWS CDK)

```
api.addChannelNamespace('lambda-direct-ns', {
  publishHandlerConfig: {
    dataSource: lambdaDataSource,
    direct: true,
  },
});
```

With this configuration, you do not need to provide code for your channel namespace. Your Lambda function will be called for every onPublish event.

Example onPublish response format

onPublish handlers that are configured with the REQUEST_RESPONSE invocation type must return a response payload with the following structure.

```
type LambdaAppSyncEventResponse = {
  events?: OutgoingEvent[], // array of outgoing events to broadcast
  error?: string //Optional error message if processing fails
}
```

Note

- If you use EVENT as the invocation type, your Lambda function is trigged asynchronously and AWS AppSync does not wait for a response. AWS AppSync broadcasts the events as usual.
- If you include an error message in the response when logging is enabled, AWS AppSync Events logs the error message but doesn't return it to the publisher.

Example onSubscribe response format

For `onSubscribe` handlers configured with `REQUEST_RESPONSE` invocation type, your Lambda function must return one of the following.

- A payload containing an error message
- `null` to indicate a successful subscription

```
type LambdaAppSyncEventResponse = {
  error: string // Error message if subscription fails
} | null
```

Best practices

We recommend the following best practices for using direct Lambda integrations.

- Enable logging to track error messages.
- Ensure your Lambda function handles both success and error cases.
- Test your integration with various payload scenarios.

Powertools for Lambda

You can utilize the following Powertools for Lambda to efficiently write your Lambda function handlers the following languages.

- TypeScript/Node.js
- Python
- .NET

Working with data sources for AWS AppSync Event APIs

Data sources are resources in your AWS account that AWS AppSync Events can interact with. AWS AppSync supports multiple data sources including AWS Lambda, Amazon DynamoDB, Amazon Aurora Serverless, Amazon OpenSearch Service, and HTTP endpoints. An AWS AppSync Events API can be configured to interact with multiple data sources, enabling you to process and route events efficiently.

Once you have configured a data source, you can use it as an integration in your namespaces. This allows you to interact with your data sources from your channel namespace handlers. For example, you can save events to a DynamoDB table as they are published, before they are broadcast. You can trigger a Lambda function asynchronously in response to a publish to start a processing job. You can check user information in an Amazon RDS Aurora PostgreSQL table as clients try to subscribe to a channel.

Topics

Supported data sources

AWS Lambda

Lambda functions are ideal for event processing in AWS AppSync Events. They execute in response to triggers and process events (JSON documents containing event data). Lambda functions can transform event data, perform business logic, or integrate with other AWS services. Lambda is particularly useful when you need custom event processing logic or integration with external services. In your channel namespace integrations, you can write code in your onPublish or onSubscribe handlers to invoke your Lambda function. Alternately, you can configure a direct integration without writing any code. AWS AppSync Events will directly call your Lambda function with the request context.

Amazon DynamoDB

DynamoDB provides scalable storage for event-driven applications. The core component is the table, which stores collections of data. Tables can store event data, application state, or any other

structured information. Each entry in a table is stored as an item, containing multiple attributes (fields). For example, an item might include attributes like eventId, timestamp, and payload.

DynamoDB is particularly effective for event storage and processing due to its high throughput and low latency. AWS AppSync Events can interact with DynamoDB tables to store, retrieve, and modify event data as needed.

Amazon RDS

Amazon RDS provides managed relational databases that can be used to store structured event data or application state. You can use a database instance with your preferred DB engine, such as PostgreSQL or MySQL, to maintain relational data that is relevant to your event processing workflow.

Amazon RDS is particularly useful when you need to maintain relationships between different types of events or when you need ACID compliance for your event data.

Amazon EventBridge

EventBridge complements AWS AppSync Events by providing event buses that can route events between different services. An event bus receives events from event sources and processes them based on rules. Each rule follows an event pattern that determines which events get routed to which targets.

EventBridge integration allows AWS AppSync Events to participate in broader event-driven architectures, either as a source or target of events.

Amazon OpenSearch Service

OpenSearch Service enables full-text search and analytics on your event data. You can create nodes containing indices to store and search event information. Data is stored as JSON documents within shards, enabling efficient searching and analysis of event data.

HTTP endpoints

HTTP endpoints can be used as data sources to integrate with external services or APIs. AWS AppSync Events can send HTTP requests with event data and process the responses as part of your event handling workflow. Each data source type has its own strengths and use cases in event-driven architectures. You can combine multiple data sources within a single AWS AppSync Events API to create sophisticated event processing workflows. You can also use an HTTP endpoint data source to interact with any AWS service API.

Amazon Bedrock

Amazon Bedrock enables AWS AppSync to integrate directly with foundation models for AI/ML capabilities. Through this data source, you can incorporate generative AI features into your event processing workflows without managing complex ML infrastructure. Amazon Bedrock provides access to various foundation models from leading AI companies, including Anthropic, AI21 Labs, Cohere, Meta, Stability AI, and Amazon.

Adding a data source

The following instructions describe how to add a data source to an Event API. To learn how to use event handlers to interact with your data source, see Writing event handlers.

Console

1. Sign in to the AWS Management Console and open the AppSync console.
   a. Choose your API in the **Dashboard**.
   b. In the **Sidebar**, choose **Data Sources**.
2. Choose **Create data source**.
   a. Give your data source a name. You can also give it a description, but that's optional.
   b. Choose your **Data source type**.
   c. For DynamoDB, you'll have to choose your Region, then the table in the Region. You can dictate interaction rules with your table by choosing to make a new generic table role or importing an existing role for the table. You can enable versioning, which can automatically create versions of data for each request when multiple clients are trying to update data at the same time. Versioning is used to keep and maintain multiple variants of data for conflict detection and resolution purposes. You can also enable automatic schema generation, which takes your data source and generates some of the CRUD, `List`, and `Query` operations needed to access it in your schema.

      For OpenSearch Service, you'll have to choose your Region, then the domain (cluster) in the Region. You can dictate interaction rules with your domain by choosing to make a new generic table role or importing an existing role for the table.

For Lambda, you'll have to choose your Region, then the ARN of the Lambda function in the Region. You can dictate interaction rules with your Lambda function by choosing to make a new generic table role or importing an existing role for the table.

For HTTP, you'll have to enter your HTTP endpoint.

For EventBridge, you'll have to choose your Region, then the event bus in the Region. You can dictate interaction rules with your event bus by choosing to make a new generic table role or importing an existing role for the table.

For Amazon RDS, you'll have to choose your Region, then the secret store (username and password), database name, and schema.

> **Note**
>
> If you're importing existing roles, they need a trust policy. For more information, see the IAM trust policy.

3. Choose **Create**.

> **Note**
>
> Alternatively, if you're creating a DynamoDB data source, you can go to the **Schema** page in the console, choose **Create Resources** at the top of the page, then fill out a predefined model to convert into a table. In this option, you will fill out or import the base type, configure the basic table data including the partition key, and review the schema changes.

CLI

- Create your data source by running the `create-data-source` command.

  You'll need to enter the following parameters for this command:

  1. The `api-id` of your API.
  2. The `name` of your table.

3. The `type` of data source. Depending on the data source type you choose, you might need to enter a `service-role-arn` and a `-config` tag.

An example command will look like the following:

```
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}
```

Creating an IAM trust policy for a data source

If you're using an existing IAM role for your data source, you need to grant that role the appropriate permissions to perform operations on your AWS resource, such as `PutItem` on an Amazon DynamoDB table. You also need to modify the trust policy on that role to allow AWS AppSync to use it for resource access as shown in the following example policy:

JSON

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

You can also add conditions to your trust policy to limit access to the data source as desired. Currently, `SourceArn` and `SourceAccount` keys can be used in these conditions. For example, the following policy limits access to your data source to the account `123456789012`:

JSON

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

Alternatively, you can limit access to a data source to a specific API, such as `abcdefghijklmnopq`, using the following policy:

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "appsync.amazonaws.com"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "ArnEquals": {
          "aws:SourceArn": "arn:aws:appsync:us-west-2:123456789012:apis/
abcdefghijklmnopq"
        }
      }
```

```
    }
  ]
}
```

You can limit access to all AWS AppSync APIs from a specific region, such as `us-east-1`, using the following policy:

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/*"
        }
      }
    }
  ]
}
```

Configuring authorization and authentication to secure Event APIs

AWS AppSync Events offers the following authorization types to secure Event APIs: API keys, Lambda, IAM, OpenID Connect, and Amazon Cognito user pools. Each option provides a different method of security.

1. **API Key authorization**: A simple key-based security option, with keys generated by the AppSync service.
2. **Lambda authorization**: Enables custom authorization logic, evaluated by an Lambda function .
3. **IAM authorization**: Utilizes AWS's signature version 4 signing process, allowing fine-grained access control through IAM policies.
4. **OpenID Connect authorization**: Integrates with OIDC-compliant services for user authentication.
5. **Amazon Cognito user pools**: Implements group-based access control using Amazon Cognitos user management features.

Authorization types

There are five ways you can authorize applications to interact with your AWS AppSync Event API. You specify which authorization type you use by specifying one of the following authorization type values in your AWS AppSync API or AWS CLI call:

- **`API_KEY`**

  For using API keys.
- **`AWS_LAMBDA`**

  For using an AWS Lambda function.
- **`AWS_IAM`**

  For using AWS Identity and Access Management permissions.
- **`OPENID_CONNECT`**

  For using your OpenID Connect provider.

- **`AMAZON_COGNITO_USER_POOLS`**

  For using an Amazon Cognito user pool.

When you define your API, you configure the authorization mode to connect to your Event API WebSocket. You also configure the default authorization modes to use when publishing and subscribing to messages. You can use different authorization modes for each configuration. For example, you might want your publisher to use IAM authorization for a backend process running on an Amazon EC2 instance, but you want your clients to use an API key to subscribe to messages.

Optionally, you can also configure the authorization mode to use for publishing and subscribing to messages on a namespace. When defined, these settings override the default configuration on your API. This enables you to have different settings for different namespaces on your API.

When you save changes to your API configuration, AWS AppSync starts to propagate the changes. Until your configuration change is propagated, AWS AppSync continues to serve your content from the previous configuration. After your configuration change is propagated, AWS AppSync immediately starts to serve your content based on the new configuration. While AWS AppSync is propagating your changes for an API, we can't determine whether the API is serving your content based on the previous configuration or the new configuration.

To learn more about using authorization types in your WebSocket operations, see Understanding the Event API WebSocket protocol.

To learn about publishing events using the HTTP or WebSocket endpoint, see Publishing events.

API_KEY authorization

API Keys allow unauthenticated clients to securely use your API. An API key is a hard-coded value in your application that is generated by the AWS AppSync service. You can rotate API keys from the AWS Management Console, the AWS CLI, or from the AWS AppSync API Reference.

API keys are configurable for up to 365 days, and you can extend an existing expiration date for up to another 365 days from that day.

On the client, the API key is specified by the header `x-api-key`. For example, if your `API_KEY` is `ABC123`, you can publish a message to a channel using the HTTP endpoint via curl as follows:

```
curl --location "https://YOUR_EVENT_API_ENDPOINT/event" \
```

```
--header 'Content-Type: application/json' \
--header "x-api-key:ABC123" \
--data '{
    "channel":"/news",
    "events":["\"Breaking news!\""]
}'
```

AWS_LAMBDA authorization

You can implement your own API authorization logic using an AWS Lambda function. When you use Lambda functions for authorization, the following constraint applies.

- A Lambda function authorizer must not return more than 5MB of contextual data.

For example, if your authorization token is `'ABC123'`, you can publish via curl as follows:

```
curl --location "https://YOUR_EVENT_API_ENDPOINT/event" \
--header 'Content-Type: application/json' \
--header "Authorization:ABC123" \
--data '{
    "channel":"/news",
    "events":["\"Breaking news!\""]
}'
```

Lambda functions are called before connection, publish, and subscription attempts. When caching is turned on, the return value of the function will be cached based on API ID, channel, operation and the authentication token as applicable.

> **Note**
>
> For the connect operation (EVENT_CONNECT), caching is based on API ID, operation, and the authentication token because the channel name will not be available during the connect operation. Subscribe operation (EVENT_SUBSCRIBE) allows wildcards for channel names, in which case the channel name is used as a literal string along with API ID, operation, and the authentication token to cache Lambda function's return value.

You can also specify a regular expression that validates authorization tokens before the function is called. These regular expressions are used to validate that an authorization token is of the correct

format before your function is called. Any request using a token which does not match this regular expression will be denied automatically.

Lambda functions used for authorization require a principal policy for `appsync.amazonaws.com` to be applied on them to allow AWS AppSync to call them. This action is done automatically in the AWS AppSync console; The AWS AppSync console does *not* remove the policy. For more information on attaching policies to Lambda functions, see Working with resource-based IAM policies in Lambda in the *AWS Lambda Developer Guide*.

The Lambda function you specify will receive an event with the following shape:

```
{
  "authorizationToken": "ExampleAUTHtoken123123123",
  "requestContext": {
    "apiId": "aaaaaa123123123example123",
    "accountId": "111122223333",
    "requestId": "f4081827-1111-4444-5555-5cf4695f339f",
    "operation": "EVENT_PUBLISH",
    "channelNamespaceName": "news",
    "channel": "/news/latest"
  },
  "requestHeaders": {
    "header": "value"
  }
}
```

The operation property indicates the operation that is being evaluated and can have the following values:

- `EVENT_CONNECT`

> **Note**
>
> For the `EVENT_CONNECT` operation, the `channelNamespaceName` and `channel` properties are not set and will be `NULL`.

- `EVENT_SUBSCRIBE`
- `EVENT_PUBLISH`

The event object contains the headers that were sent in the request from the application client to AWS AppSync.

The authorization function must return at least `isAuthorized`, a boolean indicating whether the request is authorized to execute the operation on the Event API.

If this value is true, execution of the Event API continues. If this value is false, an `UnauthorizedException` is raised.

Accepted keys

`isAuthorized` (boolean, required)

A boolean value indicating if the value in `authorizationToken` is authorized to execute the operation on the Event API.

If this value is true, execution of the Event API continues. If this value is false, an `UnauthorizedException` is raised

`handlerContext` (JSON object, optional)

A JSON object visible as `$ctx.identity.handlerContext` in your handlers. The object is a map of strings. For example, if the following structure is returned by a Lambda authorizer:

```
{
  "isAuthorized":true
  "handlerContext": {
    "banana":"very yellow",
    "apple":"very green"
  }
}
```

The value of `ctx.identity.handlerContext.apple` in handlers will be `very green`. The `handlerContext` object only supports key-value pairs. Nested keys are not supported.

> **Warning**
>
> The total size of this JSON object must not exceed 5MB.

`ttlOverride` (integer, optional)

The number of seconds that the response should be cached for. If no value is returned, the value from the API is used. If this is 0, the response is not cached.

Lambda authorizers have a standard timeout of 10 seconds but may time out earlier under peak traffic conditions. We recommend designing functions to execute in the shortest amount of time as possible (under 1s) to scale the performance of your API.

Multiple AWS AppSync APIs can share a single authentication Lambda function. Cross-account authorizer use is not supported.

The following example describes a Lambda function that demonstrates the various authentication and failure states that a Lambda function can have when used as an AWS AppSync authorization mechanism.

```
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,
      'handlerContext': {
        'key': 'value'
      }
    }

  # Authorized with no context
  if 'Authorized' in token:
    return {
      'isAuthorized': True
    }

  # never cache response
  if 'NeverCache' in token:
    return {
      'isAuthorized': True,
```

```
    'ttlOverride': 0
  }


# not authorized
if 'Unauthorized' in token:
  return {
    'isAuthorized': False
  }

# if nothing is returned, then the authorization fails.
return {}
```

AWS_IAM authorization

This authorization type enforces the AWS signature version 4 signing process on your API. You can associate AWS Identity and Access Management (IAM) access policies with this authorization type. Your application can leverage this association by using an access key (which consists of an access key ID and secret access key) or by using short-lived, temporary credentials provided by Amazon Cognito Federated Identities.

Use the following example if you want an IAM role that has permission to perform all data operations on an API.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "appsync:EventConnect",
                "appsync:EventPublish",
                "appsync:EventSubscribe"
            ],
            "Resource": [
                "arn:aws:appsync:us-west-2:123456789012:apis/{APIID}/*"
            ]
        }
    ]
```

```
}
```

Use the following example to restrict access to a specific API and a specific channel namespace.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "appsync:EventConnect"
            ],
            "Resource": [
            "arn:aws:appsync:us-east-1:111122223333:apis/{APIID}"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "appsync:EventPublish",
                "appsync:EventSubscribe"
            ],
            "Resource": [
            "arn:aws:appsync:us-east-1:111122223333:apis/{APIID}/
channelNamespace/{NAME}"
            ]
        }
    ]
}
```

OPENID_CONNECT authorization

This authorization type enforces OpenID connect (OIDC) tokens provided by an OIDC-compliant service. Your application can leverage users and privileges defined by your OIDC provider for controlling access.

An Issuer URL is the only required configuration value that you provide to AWS AppSync (for example, `https://auth.example.com`). This URL must be addressable over HTTPS. AWS AppSync appends `/.well-known/openid-configuration` to the issuer URL and locates the OpenID configuration at `https://auth.example.com/.well-known/openid-configuration` per the OpenID Connect Discovery specification. It expects to retrieve an RFC5785 compliant JSON document at this URL. This JSON document must contain a `jwks_uri` key, which points to the JSON Web Key Set (JWKS) document with the signing keys. AWS AppSync requires the JWKS to contain JSON fields of `kty` and `kid`.

AWS AppSync supports a wide range of signing algorithms.

Signing algorithms
RS256
RS384
RS512
PS256
PS384
PS512
HS256
HS384
HS512
ES256
ES384
ES512

We recommend that you use the RSA algorithms. Tokens issued by the provider must include the time at which the token was issued (`iat`) and may include the time at which it was authenticated (`auth_time`). You can provide TTL values for issued time (`iat`TTL) and authentication time

(authTTL) in your OpenID Connect configuration for additional validation. If your provider authorizes multiple applications, you can also provide a regular expression (clientId) that is used to authorize by client ID. When the clientId is present in your OpenID Connect configuration, AWS AppSync validates the claim by requiring the clientId to match with either the aud or azp claim in the token.

To validate multiple client IDs use the pipeline operator ("|") which is an "or" in regular expression. For example, if your OIDC application has four clients with client IDs such as 0A1S2D, 1F4G9H, 1J6L4B, 6GS5MG, to validate only the first three client IDs, you would place 1F4G9H|1J6L4B|6GS5MG in the client ID field.

AMAZON_COGNITO_USER_POOLS authorization

This authorization type enforces OIDC tokens provided by Amazon Cognito user pools. Your application can leverage the users and groups in your handlers to apply custom business rules.

When using Amazon Cognito user pools, you can create groups that users belong to. This information is encoded in a JWT token that your application sends to AWS AppSync in an authorization header with each request.

```
curl --location "https://YOUR_EVENT_API_ENDPOINT/event" \
--header 'Content-Type: application/json' \
--header "Authorization:JWT_TOKEN" \
--data '{
    "channel":"/news",
    "events":["\"Breaking news!\""]
}'
```

Circumventing SigV4 and OIDC token authorization limitations

The following methods can be used to circumvent the issue of not being able to use your SigV4 signature or OIDC token as your Lambda authorization token when certain authorization modes are enabled.

If you want to use the SigV4 signature as the Lambda authorization token when the AWS_IAM and AWS_LAMBDA authorization modes are enabled for AWS AppSync's API, do the following:

- To create a new Lambda authorization token, add random suffixes and/or prefixes to the SigV4 signature.

- To retrieve the original SigV4 signature, update your Lambda function by removing the random prefixes and/or suffixes from the Lambda authorization token. Then, use the original SigV4 signature for authentication.

If you want to use the OIDC token as the Lambda authorization token when the `OPENID_CONNECT` authorization mode or the `AMAZON_COGNITO_USER_POOLS` and `AWS_LAMBDA` authorization modes are enabled for AWS AppSync's API, do the following:

- To create a new Lambda authorization token, add random suffixes and/or prefixes to the OIDC token. The Lambda authorization token should not contain a Bearer scheme prefix.
- To retrieve the original OIDC token, update your Lambda function by removing the random prefixes and/or suffixes from the Lambda authorization token. Then, use the original OIDC token for authentication.

Publishing events

AWS AppSync Events allows you to publish events via your API's HTTP or WebSocket endpoint. Use the topics in this chapter to understand the publishing steps and how to structure your HTTP or WebSocket requests.

Topics

Publish events via HTTP

AWS AppSync Events allows you to publish events via your API's HTTP endpoint using a POST operation. Publishing is the only supported action over the endpoint.

Publish steps

1. Send a POST request to the address: `https://HTTP_DOMAIN/event`.
2. Add the authorization header(s) required to authorize your request.
3. Specify the following in the request body:
   - The channel that you are publishing to.
   - The list of events you are publishing. You can publish up to 5 events in a batch.

Each specified event in your publish request must be a stringified valid JSON value.

Publish example

The following is an example of a request.

```
{
  "method": "POST",
  "headers": {
    "content-type": "application/json",
    "x-api-key": "da2-your-api-key"
  },
  "body": {
```

```
    "channel": "default/channel",
    "events": [
      "{\"event_1\":\"data_1\"}",
      "{\"event_2\":\"data_2\"}"
    ]
  }
}
```

You can use your Browser's `fetch API` to publish the events. The following example demonstrates this.

```
await fetch(`https://${HTTP_DOMAIN}/event`, {
  "method": "POST",
  "headers": {
    "content-type": "application/json",
    "x-api-key": "da2-your-api-key"
  },
  "body": {
    "channel": "default/channel",
    "events": [
      "{\"event_1\":\"data_1\"}",
      "{\"event_2\":\"data_2\"}"
    ]
  }
})
```

To learn more about the different authorization types that AWS AppSync Events supports, see Configuring authorization and authentication to secure Event APIs.

Publish events via WebSocket

AWS AppSync Events allows you to publish events via your API's WebSocket endpoint after you connect to it.

Publish steps

1. Connect to your WebSocket endpoint: `wss://WS_DOMAIN/event/realtime`.
2. Send a `publish` message over the established WebSocket connection with the following information:
   - The authorization header(s) required to authorize your message.

- The following message details:
  - A unique ID for your message.
  - The channel that you are publishing to.
  - The list of events that you are publishing. You can publish a maximum of five events in a batch.

Each specified event in your publish request must be a stringified valid JSON value.

Publish example

The following is an example of a request written in JavaScript that uses an API key for authorization.

```
const message = {
  id: 'uniqueID', // Your unique id for this message.
  type: 'publish',
  channel: '/default/channel',
  events: [
    JSON.stringify({ message: "hello world" }),
    JSON.stringify({ number: 42 })
  ],
  authorization: { 'x-api-key': 'your key' },
}
```

You can use your browser's WebSocket API to connect and publish the events. The following simplified example uses an API key that sends the message. Notice that you don't need to subscribe to any channel before publishing.

```
const HTTP_DOMAIN = 'your api HTTP domain '
const REALTIME_DOMAIN = 'api real-time domain'
const API_KEY = 'your api key'

const authorization = { 'x-api-key': API_KEY, host: HTTP_DOMAIN }

// construct the protocol header for the connection
function getAuthProtocol() {
  const header = btoa(JSON.stringify(authorization))
    .replace(/\+/g, '-') // Convert '+' to '-'
    .replace(/\//g, '_') // Convert '/' to '_'
    .replace(/=+$/, '') // Remove padding `=`
```

```
  return `header-${header}`
}

const socket = await new Promise((resolve, reject) => {
  const socket = new WebSocket(`wss://${REALTIME_DOMAIN}/event/realtime`, [
    'aws-appsync-event-ws',
    getAuthProtocol(),
  ])
  socket.onopen = () => resolve(socket)
  socket.onclose = (event) => reject(new Error(event.reason))
  socket.onmessage = (event) => console.log(event)
})

// when the socket is connected, send the message
socket.send(JSON.stringify(message))
```

You receive an acknowledgement message for every publish. For more information, see Understanding the Event API WebSocket protocol.

To learn more about the different authorization types that AWS AppSync Events supports, see Configuring authorization and authentication to secure Event APIs.

Understanding the Event API WebSocket protocol

AWS AppSync Events' WebSocket API allows a client to subscribe and receive events in real-time. After the client is connected, the client can also publish to channels. Establishing a valid connection and subscribing to receive events is a simple multi-step process.

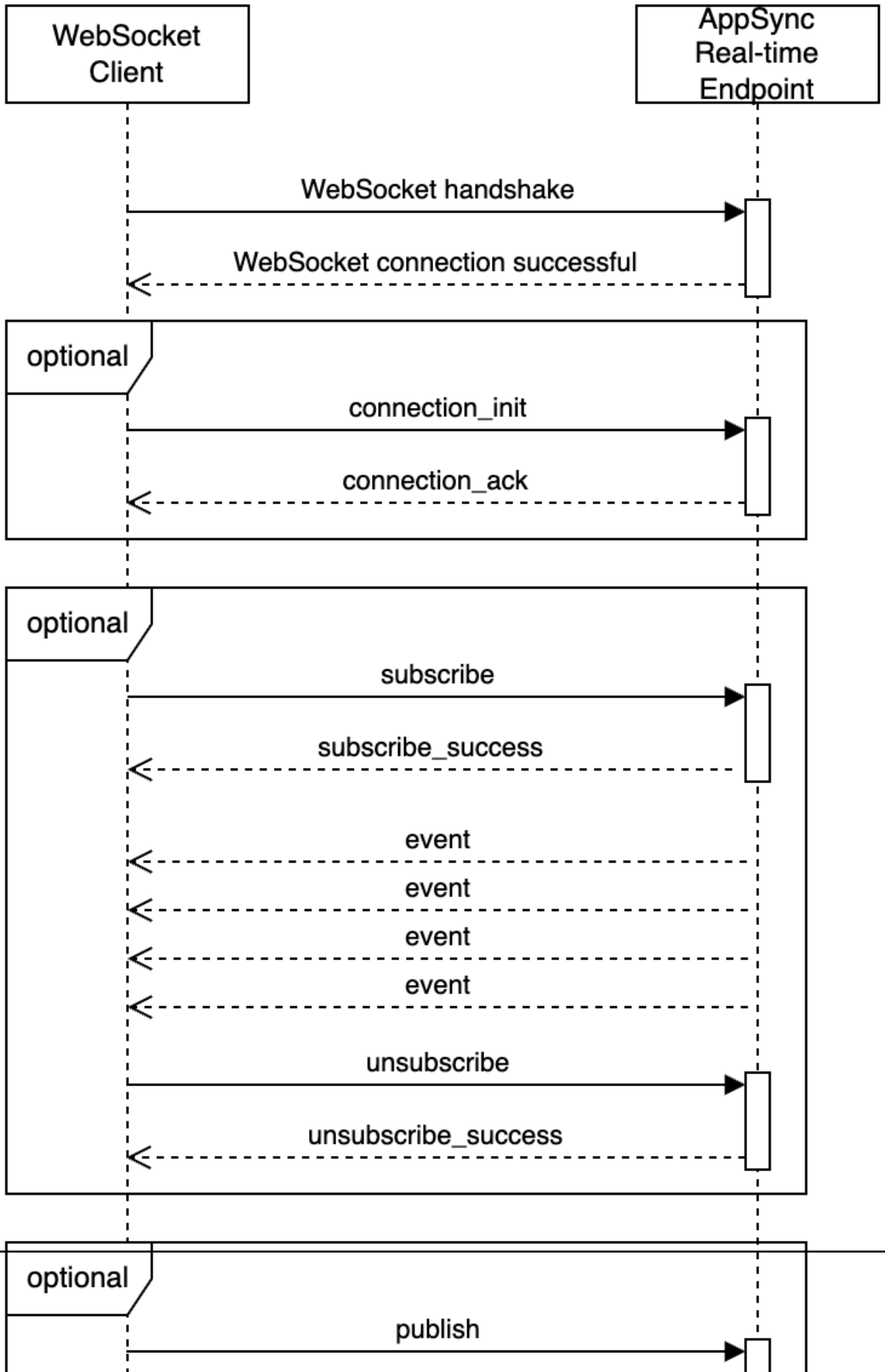
First, a client establishes a WebSocket connection with the AWS real-time endpoint, sends a connection initialization message, and waits for acknowledgment.

After a successful connection is established, the client registers subscriptions by sending a “subscribe” message with a unique ID and a channel path of interest. AWS AppSync confirms successful subscriptions with acknowledgment messages. The client then listens for subscription events, which are triggered when a publisher publishes events that are broadcast by the service. To maintain the connection, AWS AppSync sends periodic keep-alive messages.

When finished, the client unsubscribes by sending “unsubscribe” messages. This system supports multiple subscriptions on a single WebSocket connection and accommodates various authorization modes, including API keys, Amazon Cognito user pools, IAM, and Lambda.

The following diagram demonstrates the WebSocket protocol message flow between the WebSocket client and the real-time endpoint.

WebSocket protocol overview


WebSocket Client
AppSync Real-time Endpoint
WebSocket handshake
WebSocket connection successful
optional
connection_init
connection_ack
optional
subscribe
subscribe_success
event
event
event
event
unsubscribe
unsubscribe_success
optional
publish

In the preceeding diagram, the following WebSocket steps occur in the message flow.

- A client establishes a WebSocket connection with the AWS AppSync real-time endpoint. If there is a network error, the client should do a jittered exponential backoff. For more information, see Exponential backoff and jitter on the *AWS Architecture Blog*.

- After successfully establishing the WebSocket connection, the client can optionally send a `connection_init` message.
  - The client waits for a `connection_ack` message from AWS AppSync. This message includes a `connectionTimeoutMs` parameter, which is the maximum wait time in milliseconds for a "ka" (keep-alive) message.

- AWS AppSync sends "ka" messages periodically. The client keeps track of the time that it received each "ka" message. If the client doesn't receive a "ka" message within `connectionTimeoutMs` milliseconds, the client should close the connection.

- The client registers the subscription by sending a subscribe message. A single WebSocket connection supports multiple subscriptions, even if they are in different authorization modes.

- The client waits for AWS AppSync to send `subscribe_success` messages to confirm successful subscriptions.

- The client listens for subscription events, which are sent after events are published to the channel of interest.

- The client unregisters the subscription by sending an `unsubscribe` subscription message.
- At any time after connection, the client can publish a message by sending a `publish` message. Subscribing to a channel is not required before publishing.
- The client receives a `publish_success` acknowledgement.

- After unregistering all subscriptions and checking that there are no messages transferring through the WebSocket, the client can disconnect from the WebSocket connection.

Handshake details to establish the WebSocket connection

All interactions with the AWS AppSync real-time endpoint begin with establishing a WebSocket connection. The connection remains open as long as the client remains connected, up to a maximum of 24 hours. Connecting is an operation that requires authorization credentials to complete the handshake. To connect and initiate a successful handshake with AWS AppSync, a WebSocket client needs the following information:

- The AWS AppSync Events realtime and HTTP endpoints
- The authorization details

To authorize your WebSocket connection establishment, send the authorization information as a WebSocket subprotocol. To do this, a client must wrap the appropriate authorization credentials in a JSON object, encode the object in Base64URL format, and append the encoded header string in the list of subprotocols.

The following JavaScript example converts an authorization object into a base64URL encoded string.

```
/**
 * Encodes an object into Base64 URL format
 * @param {*} authorization - an object with the required authorization properties
 **/
function getBase64URLEncoded(authorization) {
  return btoa(JSON.stringify(authorization))
    .replace(/\+/g, '-') // Convert '+' to '-'
    .replace(/\//g, '_') // Convert '/' to '_'
    .replace(/=+$/, '') // Remove padding `=`
}
```

Next, this example creates the required subprotocol value.

```
function getAuthProtocol(authorization) {
  const header = getBase64URLEncoded(authorization)
  return `header-${header}`
}
```

The following example uses bash to create a header, and then uses wscat to connect. You must specify `aws-appsync-event-ws` as one of the subprotocols.

```
$ REALTIME_DOMAIN='example1234567890000.appsync-realtime-api.us-west-2.amazonaws.com'
$ HTTP_DOMAIN='example1234567890000.appsync-api.us-east-1.amazonaws.com'
$ API_KEY='da2-12345678901234567890123456'

$ header="{\"host\":\"$HTTP_DOMAIN\", \"x-api-key\":\"$API_KEY\"}"
$ header=`echo "$header" | base64 | tr '+/' '-_' | tr -d '\n='`
$ wscat -p 13 -s "header-$header" -s "aws-appsync-event-ws" -c "wss://$REALTIME_DOMAIN/
event/realtime"

Connected (press CTRL+C to quit)
```

Discovering the real-time endpoint from the Event API endpoint

AWS AppSync Event APIs are configured with two endpoints: a realtime endpoint and an HTTP endpoint. You can retrieve your endpoint information by visiting your API's **Settings** page in the AWS Management Console or by running the AWS CLI command `aws appsync get-api`.

AWS AppSync Events HTTP endpoint

https://example1234567890000.appsync-api.us-east-1.amazonaws.com/event

AWS AppSync Events real-time endpoint

wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/event/realtime

Applications can connect to the HTTP endpoint (https://) using any HTTP client, and can connect to the real-time endpoint (wss://) using any WebSocket client.

With custom domain names, you can interact with both endpoints using a single domain. For example, if you configure api.example.com as your custom domain, you can interact with your HTTP and real-time endpoints using the following URLs.

AWS AppSync Events HTTP endpoint

https://api.example.com/event

AWS AppSync Events real-time endpoint

wss://api.example.com/event/realtime

Authorization formatting based on the AWS AppSync API authorization mode

The format of the authorization subprotocol varies depending on the AWS AppSync authorization mode. AWS AppSync supports API key, Amazon Cognito user pools, OpenID Connect (OIDC), AWS Lambda, and IAM authorization modes. The host field in the object refers to the AWS AppSync Events HTTP endpoint, which is used to validate the connection even if the wss:// call is made against the real-time endpoint.

Use the following sections to learn how to format the authorization subprotocol for the supported authorization modes.

API key subprotocol format

Header content

- `"host": <string>`: The host for the AWS AppSync Events HTTP endpoint or your custom domain name. Only required for connection authorization.
- `"x-api-key": <string>`: The API key configured for the AWS AppSync Event API.

Example

```
{
  "host":"example1234567890000.appsync-api.us-east-1.amazonaws.com",
  "x-api-key":"da2-12345678901234567890123456"
}
```

Amazon Cognito user pools and OpenID Connect (OIDC) subprotocol format

Header content

- `"host": <string>`: The host for the AWS AppSync Events HTTP endpoint or your custom domain name. Only required for connection authorization.
- `"Authorization": <string>`: A JWT ID token. The header can use a Bearer scheme.

Example

```
{

 "Authorization":"eyEXAMPLEiJjbG5xb3A5eW5MK09QYXIrMTJHWEFLSXBieU5WNHhsQjEXAMPLEnM2WldvPSIsImFsZ
zEE2DJH7sHOl2zxYi7f-SmEGoh2AD8emxQRYajByz-rE4Jh0QOymN2Ys-ZIkMpVBTPgu-
TMWDyOHhDUmUj2OP82yeZ3wlZAtr_gM4LzjXUXmI_K2yGjuXfXTaa1mvQEBG0mQfVd7SfwXB-
jcv4RYVi6j25qgow9Ew52ufurPqaK-3WAKG32KpV8J4-Wejq8t0c-
yA7sb8EnB551b7TU93uKRiVVK3E55Nk5ADPoam_WYE45i3s5qVAP_-InW75NUoOCGTsS8YWMfb6ecHYJ-1j-
bzA27zaT9VjctXn9byNFZmEXAMPLExw",
  "host":"example1234567890000.appsync-api.us-east-1.amazonaws.com"
}
```

AWS Lambda subprotocol format

Header content

- `"host": <string>`: The host for the AWS AppSync Events HTTP endpoint or your custom domain name. Only required for connection authorization.
- `"Authorization": <string>`: A custom authorization token of your design.

Example

```
{

 "Authorization":"M0UzQzM1MkQtMkI0Ni00OTZCLUI1NkQtMUM0MTQ0QjVBRTczCkI1REEzRTIxLTk5NzItNDJENi1BQ
  "host":"example1234567890000.appsync-api.us-east-1.amazonaws.com"
}
```

AWS Identity and Access Management (IAM) subprotocol format

Header content

- `"accept": "application/json, text/javascript"`: A constant string parameter.
- `"content-encoding": "amz-1.0"`: A constant string parameter.
- `"content-type": "application/json; charset=UTF-8"`: A constant string parameter.
- `"host": <string>`: This is the host for the AWS AppSync Events HTTP endpoint.
- `"x-amz-date": <string>`: The timestamp must be in UTC and in the following ISO 8601 format: YYYYMMDD'T'HHMMSS'Z'. For example, 20150830T123600Z is a valid timestamp.

Don't include milliseconds in the timestamp. For more information, see Elements of an AWS API request signature in the *IAM User Guide*.

- `"X-Amz-Security-Token": <string>`: The AWS session token, which is required when using temporary security credentials. For more information, see Use temporary credentials with AWS resources in the *IAM User Guide*.
- `"Authorization": <string>`: Signature Version 4 (SigV4) signing information for the AWS AppSync endpoint. For more information on the signing process, see Create a signed AWS API request in the *IAM User Guide*.

The SigV4 signing HTTP request includes a canonical URL, which is the AWS AppSync HTTP endpoint with `/event` appended. The service endpoint AWS Region is the same Region where you're using the AWS AppSync API, and the service name is 'appsync'.

The HTTP request to sign to connect is the following.

```
{
  url: "https://example1234567890000.appsync-api.us-east-1.amazonaws.com/event",
  data: "{}",
  method: "POST",
  headers: {
    "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",
  }
}
```

The following is the request to sign when sending a subscribe message. The channel name is specified in the request.

```
{
  url: "https://example1234567890000.appsync-api.us-east-1.amazonaws.com/event",
  body: "{\"channel\":\"/your/channel/*\"}",
  method: "POST",
  headers: {
    "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",
  }
```

```
}
```

The following is the request to sign when publishing over WebSocket. The channel name and event payload are specified in the request.

```
{
  url: "https://example1234567890000.appsync-api.us-east-1.amazonaws.com/event",
  body: "{\"channel\":\"/your/channel/*\",\"events\":[{\"event_1\":\"data_1\"},
{\"event_2\":\"data_2\"}]}",
  method: "POST",
  headers: {
    "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",
  }
}
```

Authorization header example

```
{
  "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":
 "AgEXAMPLEZ2luX2VjEAoaDmFwLXNvdXRoZWFEXAMPLEcwRQIgAh97Cljq7wOPL8KsxP3YtDuyc/9hAj8PhJ7Fvf38SgoC
+
+pEagWCveZUjKEn0zyUhBEXAMPLEjj//////////8BEXAMPLExODk2NDgyNzg1NSIMo1mWnpESWUoYw4BkKqEFSrm3DXuL8
+ZbVc4JKjDP4vUCKNR6Le9C9pZp9PsW0NoFy3vLBUdAXEXAMPLEOVG8feXfiEEA+1khgFK/
wEtwR+9zF7NaMMMse07wN2gG2tH0eKMEXAMPLEQX+sMbytQo8iepP9PZOzlZsSFb/
dP5Q8hk6YEXAMPLEYcKZsTkDAq2uKFQ8mYUVA9EtQnNRiFLEY83aKvG/tqLWNnGlSNVx7SMcfovkFDqQamm
+88y1OwwAEYK7qcoceX6Z7GGcaYuIfGpaX2MCCELeQvZ+8WxEgOnIfz7GYvsYNjLZSaRnV4G
+ILY1F0QNW64S9Nvj
+BwDg3ht2CrNvpwjVYlj9U3nmxE0UG5ne83LL5hhqMpm25kmL7enVgw2kQzmU2id4IKu0C/
WaoDRuO2F5zE63vJbxN8AYs7338+4B4HBb6BZ6OUgg96Q15RA41/
gIqxaVPxyTpDfTU5GfSLxocdYeniqqpFMtZG2n9d0u7GsQNcFkNcG3qDZm4tDo8tZbuym0a2VcF2E5hFEgXBa
+XLJCfXi/77OqAEjP0x7Qdk3B43p8KG/BaioP5RsV8zBGvH1zAgyPha2rN70/
tT13yrmPd5QYEfwzexjKrV4mWIuRg8NTHYSZJUaeyCwTom80VFUJXG
+GYTUyv5W22aBcnoRGiCiKEYTLOkgXecdKFTHmcIAejQ9Welr0a196Kq87w5KNMCkcCGFnwBNFLmfnbpNqT6rUBxxs3X5nt
aox0FtHX21eF6qIGT8j1z+l2opU+ggwUgkhUUgCH2TfqBj+MLMVVvpgqJsPKt582caFKArIFIvO
+9QupxLnEH2hz04TMTfnU6bQC6z1buVe7h
```

```
+tOLnh1YPFsLQ88anib/7TTC8k9DsBTq0ASe8R2GbSEsmO9qbbMwgEaYUhOKtGeyQsSJdhSk6XxXThrWL9EnwBCXDkICMqd
+WgtPtKOOweDlCaRs3R2qXcbNgVhleMk4IWnF8D1695AenU1LwHjOJLkCjxgNFiWAFEPH9aEXAMPLExA==",
  "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"
}
```

Real-time WebSocket operations

After initiating a successful WebSocket handshake with AWS AppSync, the client must send a subsequent message to connect to AWS AppSync for different operations. The WebSocket API has the following properties.

WebSocket API message properties

id

The client provided ID of the operation. This property is required and is used to correlate response error and success messages. For subscriptions, this property must be unique for all subscriptions within a connection. The property is a string and is limited to a maximum of 128 alphanumeric and special character (`_,+,-`) characters: `/^[a-zA-Z0-9-_+]{1,128}$/`.

type

The type of operation being performed. Supported client operations are subscribe, unsubscribe, publish. The property is a string and must be one of the message types defined in the next section, *Configuring message details*.

channel

The channel to subscribe to, or to publish events to. The property is a string made up of one to five segments separated by a slash. Each segment is limited to 50 alphanumeric + dash characters. The property is case sensitive. For example: `channelNamespaceName` or `channelNamespaceName/sub-segment-1/subSegment-2` `/^\/?[A-Za-z0-9](?:[A-Za-z0-9-]{0,48}[A-Za-z0-9])?(?:\/[A-Za-z0-9](?:[A-Za-z0-9-]{0,48}[A-Za-z0-9])?){0,4}\/?$/`

events

An array of events to be published. You can publish up to five events in a batch. Each specified event in your publish request must be a stringified valid JSON value.

authorization

The authorization headers necessary to authorize the operation. For example, ApiKey will contain `x-api-key`, while Amazon Cognito, OIDC, and Lambda will contain `authorization`. IAM will contain `host`, `x-amz-date`, `x-amz-security-token`, and `authorization`. The `host` header is required for IAM only.

Configuring message details

This section provides information about the syntax to use to configure the details for various message types.

Connection init message

(Optional) After the client has established the WebSocket connection, the client sends an init message to initiate the connection session.

```
{ "type": "connection_init" }
```

Connection acknowledge message

AWS AppSync responds with an “ack” message that contains a connection timeout value. If the client doesn’t receive a keep-alive message within the connection timeout period, the client should close the connection. The connection timeout period is 5 minutes.

```
{
    "type": "connection_ack",
    "connectionTimeoutMs": 300000
}
```

Keep-alive message

AWS AppSync periodically sends a keep-alive message to the client to maintain the connection. If the client doesn’t receive a keep-alive message within the connection timeout period, the client should close the connection. The keep-alive interval is 60 seconds. Clients do not need to acknowledge these messages.

```
{ "type": "ka" }
```

Subscribe message

After receiving a `connection_ack` message, the client can send a subscription registration message to listen for events on a channel.

- “id” is the ID of the subscription. This ID must be unique per client connection otherwise AWS AppSync returns an error message indicating the subscription message is duplicated.
- “channel” is the channel to which the subscribed client is listening. Any messages published to this channel will be delivered to the subscribed client.
- “authorization” is an object containing the fields required for authorization. The authorization object follows the same rules as the headers for connecting to the WebSocket.

```
{
  "type": "subscribe",
  "id": "ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69",
  "channel": "/namespaceA/subB/subC",
  "authorization": {
    "x-api-key": "da2-12345678901234567890123456",
    "host": "example1234567890000.appsync-api.us-east-1.amazonaws.com"
   }
}
```

Subscription acknowledgment message

AWS AppSync acknowledges with a success message. “id” is the ID of the corresponding subscribe operation that succeeded.

```
{
  "type": "subscribe_success",
  "id": "ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69"
}
```

In case of an error, AWS AppSync sends a `subscribe_error` response.

```
{
  "type": "subscribe_error",
  "id": "ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69",
  "errors": [
    {
```

```
      "errorType": "SubscriptionProcessingError",
      "message": "There was an error processing the operation"
    }
  ]
}
```

Data message

When an event is published to a channel the client is subscribed to, the event is broadcast and delivered in a data message. “id” is the ID of the corresponding subscription for the channel to which the message was published.

```
{
  "type": "data",
  "id": "ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69",
  "event": ["\"my event content\""]
}
```

In case of an error, such as a broadcasting error, an error can be received at the client:

```
{
  "type": "broadcast_error",
  "id": "ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69",
  "errors": [
    {
      "errorType": "MessageProcessingError",
      "message": "There was an error processing the message"
    }
  ]
}
```

Unsubscribe message

When the client wants to stop listening to a subscribed channel, the client sends a message to unregister the subscription. “id” is the ID of the corresponding subscription to which the client wants to unregister.

```
{
  "type": "unsubscribe",
  "id": "ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69"
}
```

AWS AppSync acknowledges with a success message. “id” is the ID of the corresponding subscribe operation that succeeded.

```
{
  "type": "unsubscribe_success",
  "id": "ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69"
}
```

If an error occurs, an error message is sent back to the client

```
{
  "type": "unsubscribe_error",
  "id": "ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69",
  "errors": [
    {
      "errorType": "UnknownOperationError",
      "message": "Unknown operation id ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69"
    }
  ]
}
```

Publish message

Once connected, the client can start publishing messages to channels. “`id`” is the ID of the publish operation. “`channel`” is the channel to which the events are published. The events will be delivered to all clients subscribed to the channel. “`authorization`” is an object containing the fields required for authorization. The authorization object follows the same rules as the headers for subscribing to the channel.

```
{
  "type": "publish",
  "id": "ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69",
  "channel": "/namespaceA/subB/subC",
  "events": [ "{ \"msg\": \"Hello World!\" }" ],
  "authorization": {
    "x-api-key": "da2-12345678901234567890123456",
   }
}
```

Publish success message

The server acknowledges with a success message. “`id`” is the ID of the corresponding publish operation that succeeded. “`failed`” is the list of events which were marked in error while executing the publish handler.

```
{
  "id": "ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69",
  "type": "publish_success",
  "successful": [
    {
      "identifier": "221b6b3f-46fd-4ac1-a327-bd2623b7402e",
      "index": 0
    }
  ],
  "failed": []
}
```

If an error occurs, an error message is sent back to the client.

```
{
    "id": "ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69",
    "type": "publish_error",
    "errors": [
        {
            "errorType": "An error type",
            "message": "A message"
        }
    ]
}
```

Disconnecting the WebSocket

Before disconnecting the WebSocket, to avoid data loss, the client should have the necessary logic to check that no operation is currently in place through the WebSocket connection. All subscriptions should be unregistered before disconnecting from the WebSocket.

Configuring custom domain names for Event APIs

With AWS AppSync, you can use custom domain names to configure a single, memorable domain that works for your Event APIs.

When you configure an AWS AppSync Event API, two endpoints are provisioned: An HTTP endpoint and a real-time endpoint. These endpoints have the following format.

AWS AppSync Events HTTP endpoint

https://example1234567890000.appsync-api.us-east-1.amazonaws.com/event

AWS AppSync Events real-time endpoint

wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/event/realtime

With custom domain names, you can interact with both endpoints using a single domain. For example, if you configure api.example.com as your custom domain, you can interact with both your HTTP and real-time WebSocket endpoints using the following URLs.

AWS AppSync Events HTTP endpoint

https://api.example.com/event

AWS AppSync Events real-time endpoint

wss://api.example.com/event/realtime

> **Note**
>
> AWS AppSync APIs support only TLS 1.2 and TLS 1.3 for custom domain names.

Registering and configuring a domain name for an Event API

To set up custom domain names for your AWS AppSync APIs, you must have a registered internet domain name. You can register an internet domain using Amazon Route 53 domain registration or a third-party domain registrar of your choice. For more information about using Route 53, see What is Amazon Route 53 in the *Amazon Route 53 Developer Guide*.

An API's custom domain name can be the name of a subdomain or the root domain (also known as the "zone apex") of a registered internet domain. After you create a custom domain name in AWS AppSync, you must create or update your DNS provider's resource record to map to your API endpoint. Without this mapping, API requests bound for the custom domain name cannot reach AWS AppSync.

Creating a custom domain name in AWS AppSync

Creating a custom domain name for an AWS AppSync API sets up an Amazon CloudFront distribution. You must set up a DNS record to map the custom domain name to the CloudFront distribution domain name. This mapping is required to route API requests that are bound for the custom domain name in AWS AppSync through the mapped CloudFront distribution.

You must also provide a certificate for the custom domain name. To set up the custom domain name or to update its certificate, you must have permission to update CloudFront distributions and describe the AWS Certificate Manager (ACM) certificate that you plan to use. To grant these permissions, attach the following AWS Identity and Access Management (IAM) policy statement to an IAM user, group, or role in your 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 supports custom domain names by leveraging Server Name Indication (SNI) on the CloudFront distribution. For more information about using custom domain names on a CloudFront distribution, including the required certificate format and the maximum certificate key length, see Using HTTPS with CloudFront in the *Amazon CloudFront Developer Guide*.

To set up a custom domain name as the API's hostname, the API owner must provide an SSL/TLS certificate for the custom domain name. To provide a certificate, do one of the following.

- Request a new certificate in ACM, or import a certificate issued by a third-party certificate authority into ACM in the US East (N. Virginia) (us-east-1) AWS Region. For more information about ACM, see What is AWS Certificate Manager in the *AWS Certificate Manager User Guide*.
- Provide an IAM server certificate. For more information, see Manage server certificates in IAM in the *IAM User Guide*.

Wildcard custom domain names in AWS AppSync

AWS AppSync supports wildcard custom domain names. To configure a wildcard custom domain name, specify a wildcard character (*) as the first subdomain of a custom domain. This represents all possible subdomains of the root domain. For example, the wildcard custom domain name `*.example.com` results in subdomains such as `a.example.com`, `b.example.com`, and `c.example.com`. All these subdomains route to the same domain.

To use a wildcard custom domain name in AWS AppSync, you must provide a certificate issued by ACM containing a wildcard name that can protect several sites in the same domain. For more information, see ACM certificate characteristics and limitations in the *AWS Certificate Manager User Guide*.

Using CloudWatch to monitor and log Event API data

You can log and debug your Event API using CloudWatch metrics and CloudWatch logs. These tools enable developers to monitor performance, troubleshoot issues, and optimize their AWS AppSync API operations effectively.

CloudWatch metrics is a tool that provides a wide range of metrics to monitor API performance and usage. These metrics fall into two main categories:

1. **HTTP API Metrics for Publish**: These include `4XXError` and `5XXError` for tracking client and server errors, `Latency` for measuring response times, `Requests` for monitoring total API calls, `TokensConsumed` for tracking resource usage, and *Events* related to metrics for tracking event publishing performance.
2. **Real-time Subscription Metrics**: These metrics focus on WebSocket connections and subscription activities. They include metrics for connection requests, successful connections, subscription registrations, message publishing, and active connections and subscriptions.

CloudWatch Logs is a tool that enables logging capabilities for your Eent APIs. Logs can be set at two levels of the API:

1. **Request-level Logs**: These capture overall request information, including HTTP headers, operation summaries, and subscription registrations.
2. **Handler-level Logs**: These provide detailed information about handler evaluation, including request and response mappings, and tracing information for each field.

You can configure logging, interpret log entries, and use log data for troubleshooting and optimization. AWS AppSync provides various log types that provide insight into your API's behavior.

Setting up and configuring logging on an Event API

Us the following instruction to turn on automatic logging on a Event API using the AWS AppSync console.

1. Sign in to the AWS Management Console and open the AppSync console.
2. On the **APIs** page, choose the name of an Event API.

3. On the API's homepage, in the navigation pane, choose **Settings**.
4. Under **Logging**, do the following:

   a. Turn on **Enable Logs**.

   b. (Optional) For **Log level**, choose your preferred field-level logging level (**None**, **Error**, or **All**).

   c. The procedure for adding a service role varies depending on whether you want to create a new role or use an existing one.

   - To create a new role:

     - For **Create or use an existing role**, choose **New role**. This creates a new IAM role that allows AWS AppSync to write logs to CloudWatch.

   - To use an existing role:

     i. Choose **Existing role**.

     ii. In the service role list, select the ARN of an existing IAM role in your AWS account.

     For information about the configuration of the IAM role, see Manually creating an IAM role with CloudWatch Logs permissions.

5. Choose **Save**.

Manually creating an IAM role with CloudWatch Logs permissions

If you choose to use an existing IAM role, the role must grant AWS AppSync the required permissions to write logs to CloudWatch. To configure this manually, you must provide a service role ARN so that AWS AppSync can assume the role when writing the logs.

In the IAM console, create a new policy with the name `AWSAppSyncPushToCloudWatchLogsPolicy` that has the following definition:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
```

```
            "Action": [
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:PutLogEvents"
            ],
            "Resource": "*"
        }
    ]
}
```

Next, create a new role with the name **AWSAppSyncPushToCloudWatchLogsRole**, and attach the newly created policy to the role. Edit the trust relationship for this role to the following:

JSON

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

Copy the role ARN and use it when setting up logging for an AWS AppSync Event API.

CloudWatch metrics

You can use CloudWatch metrics to monitor and provide alerts about specific events that can result in HTTP status codes or from latency.

HTTP endpoint metrics

`4XXError`

Errors resulting from requests that are not valid due to an incorrect client configuration. For example, these errors can occur when the request includes an incorrect JSON payload, when the service is throttled, or when the authorization settings are misconfigured.

Unit: *Count*. Use the Sum statistic to get the total occurrences of these errors.

`5XXError`

Errors encountered during the execution of a request. This could also happen if AWS AppSyncc encounters an issue during processing of a request.

Unit: *Count*. Use the Sum statistic to get the total occurrences of these errors.

`Latency`

The time between when AWS AppSync receives a request from a client and when it returns a response to the client. This doesn't include the network latency encountered for a response to reach the end devices.

Unit: *Millisecond*. Use the Average statistic to evaluate expected latencies.

`Requests`

The number of requests that all APIs in your account have processed, by Region.

Unit: *Count*. The number of all requests processed in a particular Region.

`TokensConsumed`

Tokens are allocated to `Requests` based on the amount of resources (processing time and memory used) that a `Request` consumes. Usually, each `Request` consumes one token. However, a `Request` that consumes large amounts of resources is allocated additional tokens as needed.

Unit: *Count*. The number of tokens allocated to requests processed in a particular Region.

Handler metrics

`DroppedEvents`

The count of input events filtered by a OnPublish handler.

Unit: *Count*.

`FailedEvents`

The count of input events that encountered error during processing.

Unit: *Count*.

`SuccessfulEvents`

The count of input events that were processed successfully and submitted for broadcast in the OnPublish handler.

Unit: *Count*.

`PublishedHandlerInvocations`

The number of OnPublish handler invocations.

Unit: *Count*.

Real-time endpoint metrics

`ConnectRequests`

The number of WebSocket connection requests made to AWS AppSync, including both successful and unsuccessful attempts.

Unit: *Count*. Use the Sum statistic to get the total number of connection requests.

`ConnectSuccess`

The number of successful WebSocket connections to AWS AppSync. It is possible to have connections without subscriptions.

Unit: *Count*. Use the Sum statistic to get the total occurrences of the successful connections.

`ConnectClientError`

The number of WebSocket connections that were rejected by AWS AppSync because of client-side errors. This could imply that the service is throttled or that the authorization settings are misconfigured.

Unit: *Count*. Use the Sum statistic to get the total occurrences of the client-side connection errors.

`ConnectServerError`

The number of errors that originated from AWS AppSync while processing connections. This usually happens when an unexpected server-side issue occurs.

Unit: *Count*. Use the Sum statistic to get the total occurrences of the server-side connection errors.

`DisconnectSuccess`

The number of successful WebSocket disconnections from AWS AppSync.

Unit: *Count*. Use the Sum statistic to get the total occurrences of the successful disconnections.

`DisconnectClientError`

The number of client errors that originated from AWS AppSync while disconnecting WebSocket connections.

Unit: *Count*. Use the Sum statistic to get the total occurrences of the disconnection errors.

`DisconnectServerError`

The number of server errors that originated from AWS AppSync while disconnecting WebSocket connections.

Unit: *Count*. Use the Sum statistic to get the total occurrences of the disconnection errors.

`SubscribeSuccess`

The number of subscriptions that were successfully registered to AWS AppSync through WebSocket. It's possible to have connections without subscriptions, but it's not possible to have subscriptions without connections.

Unit: *Count*. Use the Sum statistic to get the total occurrences of the successful subscriptions.

`SubscribeClientError`

The number of subscriptions that were rejected by AWS AppSync because of client-side errors. This can occur when a JSON payload is incorrect, the service is throttled, or the authorization settings are misconfigured.

Unit: *Count*. Use the Sum statistic to get the total occurrences of the client-side subscription errors.

`SubscribeServerError`

The number of errors that originated from AWS AppSync while processing subscriptions. This usually happens when an unexpected server-side issue occurs.

Unit: *Count*. Use the Sum statistic to get the total occurrences of the server-side subscription errors.

`UnsubscribeSuccess`

The number of unsubscribe requests that were successfully processed.

Unit: *Count*. Use the Sum statistic to get the total occurrences of the successful unsubscribe requests.

`UnsubscribeClientError`

The number of unsubscribe requests that were rejected by AWS AppSync because of client-side errors.

Unit: *Count*. Use the Sum statistic to get the total occurrences of the client-side unsubscribe request errors.

`UnsubscribeServerError`

The number of errors that originated from AWS AppSync while processing unsubscribe requests. This usually happens when an unexpected server-side issue occurs.

Unit: *Count*. Use the Sum statistic to get the total occurrences of the server-side unsubscribe request errors.

`BroadcastEventSuccess`

The number of events that were successfully broadcast to subscribers.

Unit: *Count*. Use the Sum statistic to get the total of events that were successfully broadcast.

`BroadcastEventClientError`

The number of events that failed to broadcast because of client-side errors.

`Unit`: *Count*. Use the Sum statistic to get the total occurrences of the client-side broadcast events errors

`BroadcastEventServerError`

The number of errors that originated from AWS AppSync while broadcasting events . This usually happens when an unexpected server-side issue occurs.

Unit: *Count*. Use the Sum statistic to get the total occurrences of the server-side broadcast errors.

`BroadcastEventSize`

The size of events broadcast.

Unit: *Bytes*.

`ActiveConnections`

The number of concurrent WebSocket connections from clients to AWS AppSync in 1 minute.

Unit: *Count*. Use the Sum statistic to get the total opened connections.

`ActiveSubscriptions`

The number of concurrent subscriptions from clients in 1 minute.

Unit: *Count*. Use the Sum statistic to get the total active subscriptions.

`ConnectionDuration`

The amount of time that the connection stays open.

Unit: *Milliseconds*. Use the Average statistic to evaluate connection duration.

`InboundMessages`

The number of inbound metered events. One metered event equals 5 kB of received event.

Unit: *Count*.

`OutboundMessages`

The number of metered messages successfully published. One metered message equals 5 kB of delivered data.

Unit: *Count*. Use the Sum statistic to get the total number of successfully published metered messages.

`InboundMessageDelayed`

The number of delayed inbound messages. Inbound messages can be delayed when either the inbound message rate quota or outbound message rate quota is breached.

Unit: *Count*. Use the Sum statistic to get the total number of inbound messages that were delayed.

`InboundMessageDropped`

The number of delayed inbound messages. Inbound messages can be delayed when either the inbound message rate quota or outbound message rate quota is breached.

Unit: *Count*. Use the Sum statistic to get the total number of inbound messages that were dropped.

`SubscribeHandlerInvocations`

The number of Subscribe handlers invoked.

Unit: *Count*.

`PublishSuccess`

The number of publish requests that were successfully sent on a WebSocket connection to AWS AppSync.

Unit: *Count*. Use the Sum statistic to get the total number of publish requests that were successfully sent.

`PublishClientError`

The number of publish requests that were rejected by AWS AppSync because of client-side errors when sending publish requests on a WebSocket connection. This can occur when a JSON payload is incorrect, the service is throttled, or the authorization settings are misconfigured.

Unit: *Count*. Use the Sum statistic to get the total occurrences of these errors.

`PublishServerError`

The number of publish requests that originated from AWS AppSync while processing publish requests on a WebSocket connection. This is usually caused by an unexpected server-side issue.

Unit: *Count*. Use the Sum statistic to get the total occurrences of these errors.

Configuring CloudWatch Logs on Event APIs

You can configure two types of logging on any new or existing API: request-level logs and and handler logs.

Request-level logs

When request-level logging is configured for an API, the following information is logged.

- The number of tokens consumed
- The request and response HTTP headers
- The overall operation summary

Handler logs

When handler logging is configured for an API, the following information is logged.

- Generated request mapping with source and arguments for each field
- The transformed response mapping for each field, which includes the data as a result of resolving that field
- Tracing information for each field

If you turn on logging, AWS AppSync manages the CloudWatch Logs. The process includes creating log groups and log streams, and reporting to the log streams with these logs.

When you turn on logging on an AWS AppSync API and make requests, AWS AppSync creates a log group and log streams under the log group. The log group is named following the `/aws/appsync/apis/{api_id}` format. Within each log group, the logs are further divided into log streams. These are ordered by **Last Event Time** as logged data is reported.

Every log event is tagged with the **x-amzn-RequestId** of that request. This helps you filter log events in CloudWatch to get all logged information about that request. You can get the RequestId from the response headers of every AWS AppSync request.

The field-level logging is configured with the following log levels:

- **None - No handler logs are captured.**
- **Error - Logs the following information *only* for the fields that are in error:**
  - The error section in the server response
  - Handler errors and `console.error` logging from handlers
  - The generated request/response functions that got resolved for error fields
- **All - Logs the following information for *all* fields in the query:**
  - Custom logging from handlers
  - The generated request/response functions that got resolved for each field

Using token counts to optimize your requests

Requests that consume less than or equal to 1,500 KB-seconds of memory and vCPU time are allocated one token. Requests with resource consumption greater than 1,500 KB-seconds receive additional tokens. For example, if a request consumes 3,350 KB-seconds, AWS AppSync allocates three tokens (rounded up to the next integer value) to the request. By default, AWS AppSync allocates a maximum of 5,000 or 10,000 request tokens per second to the APIs in your account, depending upon the AWS Region in which it's deployed. If your APIs each use an average of two tokens per second, you'll be limited to 2,500 or 5,000 requests per second, respectively. If you need more tokens per second than the allotted amount, you can submit a request to increase the default quota for the rate of request tokens. For more information, see AWS AppSync endpoints and quotas in the *AWS General Reference guide* and Requesting a quota increase in the *Service Quotas User Guide*.

A high per-request token count could indicate that there's an opportunity to optimize your requests and improve the performance of your API. Factors that can increase your per-request token count include:

- The complexity of your handlers
- The amount of data returned from your handlers
- Logging configuration, and the amount of custom logging in your handlers

> **Note**
>
> In addition to AWS AppSync metrics and logs, clients can access the number of tokens consumed in a request via the response header `x-amzn-appsync-TokensConsumed`.

Log size limits

By default, if logging has been enabled, AWS AppSync will send up to 1 MB of logs per request.

Using AWS WAF to protect AWS AppSync Event APIs

AWS WAF is a web application firewall that helps protect web applications and APIs from attacks. It allows you to configure a set of rules, called a web access control list (web ACL), that allow, block, or monitor (count) web requests based on customizable web security rules and conditions that you define. When you integrate your AWS AppSync API with AWS WAF, you gain more control and visibility into the HTTP traffic accepted by your API. To learn more about AWS WAF, see How AWS WAF Works in the AWS WAF Developer Guide.

You can use AWS WAF to protect your AppSync API from common web exploits, such as SQL injection and cross-site scripting (XSS) attacks. These could affect API availability and performance, compromise security, or consume excessive resources. For example, you can create rules to allow or block requests from specified IP address ranges, requests from CIDR blocks, requests that originate from a specific country or region, requests that contain malicious SQL code, or requests that contain malicious script.

You can also create rules that match a specified string or a regular expression pattern in HTTP headers, method, query string, URI, and the request body (limited to the first 8 KB). Additionally, you can create rules to block attacks from specific user agents, bad bots, and content scrapers. For example, you can use rate-based rules to specify the number of web requests that are allowed by each client IP in a trailing, continuously updated, 5-minute period.

To learn more about the types of rules that are supported and additional AWS WAF features, see the AWS WAF Developer Guide and the AWS WAF API Reference.

> **Important**
>
> AWS WAF is your first line of defense against web exploits. When AWS WAF is enabled on an API, AWS WAF rules are evaluated before other access control features, such as API key authorization, IAM policies, OIDC tokens, and Amazon Cognito user pools.

Integrate an AppSync API with AWS WAF

You can integrate an Appsync API with AWS WAF using the AWS Management Console, the AWS CLI, AWS CloudFormation, or any other compatible client.

To integrate an AWS AppSync API with AWS WAF

1. Create an AWS WAF web ACL. For detailed steps using the AWS WAF Console, see Creating a web ACL.
2. Define the rules for the web ACL. A rule or rules are defined in the process of creating the web ACL. For information about how to structure rules, see AWS WAF rules. For examples of useful rules you can define for your AWS AppSync API, see Creating rules for a web ACL.
3. Associate the web ACL with an AWS AppSync API. You can perform this step in the AWS WAF Console or in the AppSync Console.
   - To associate the web ACL with an AWS AppSync API in the AWS WAF Console, follow the instructions for Associating or disassociating a Web ACL with an AWS resource in the AWS WAF Developer Guide.
   - To associate the web ACL with an AWS AppSync API in the AWS AppSync Console
     a. Sign in to the AWS Management Console and open the AppSync Console.
     b. Choose the API that you want to associate with a web ACL.
     c. In the navigation pane, choose **Settings**.
     d. In the **Web application firewall** section, turn on **Enable AWS WAF**.
     e. In the **Web ACL** dropdown list, choose the name of the web ACL to associate with your API.
     f. Choose **Save** to associate the web ACL with your API.

> **Note**
>
> After you create a web ACL in the AWS WAF Console, it can take a few minutes for the new web ACL to be available. If you do not see a newly created web ACL in the **Web application firewall** menu, wait a few minutes and retry the steps to associate the web ACL with your API.

> **Note**
>
> AWS WAF integration only supports the `Connect` operation for real-time endpoints. AWS AppSync will respond with an error message for any connection blocked by AWS WAF.

After you associate a web ACL with an AWS AppSync API, you will manage the web ACL using the AWS WAF APIs. You do not need to re-associate the web ACL with your AWS AppSync API unless you want to associate the AWS AppSync API with a different web ACL.

Creating rules for a web ACL

Rules define how to inspect web requests and what to do when a web request matches the inspection criteria. Rules don't exist in AWS WAF on their own. You can access a rule by name in a rule group or in the web ACL where it's defined. For more information, see AWS WAF rules. The following examples demonstrate how to define and associate rules that are useful for protecting an AppSync API.

Example web ACL rule to limit request body size

The following is an example of a rule that limits the body size of requests. This would be entered into the **Rule JSON editor** when creating a web ACL in the 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
```

```
        }
}
```

After you have created your web ACL using the preceding example rule, you must associate it with your AppSync API. As an alternative to using the AWS Management Console, you can perform this step in the AWS CLI by running the following command.

```
aws waf associate-web-acl --web-acl-id waf-web-acl-arn --resource-arn appsync-api-arn
```

It can take a few minutes for the changes to propagate, but after running this command, requests that contain a body larger than 1024 bytes will be rejected by AWS AppSync.

> **Note**
>
> After you create a new web ACL in the AWS WAF Console, it can take a few minutes for the web ACL to be available to associate with an API. If you run the CLI command and get a `WAFUnavailableEntityException` error, wait a few minutes and retry running the command.

Example web ACL rule to limit requests from a single IP address

The following is an example of a rule that throttles an AppSync API to 100 requests from a single IP address. This would be entered into the **Rule JSON editor** when creating a web ACL with a rate-based rule in the AWS WAF Console.

```
{
  "Name": "Throttle",
  "Priority": 0,
  "Action": {
    "Block": {}
  },
  "VisibilityConfig": {
    "SampledRequestsEnabled": true,
    "CloudWatchMetricsEnabled": true,
    "MetricName": "Throttle"
  },
  "Statement": {
    "RateBasedStatement": {
      "Limit": 100,
```

```
      "AggregateKeyType": "IP"
    }
  }
}
```

After you have created your web ACL using the preceding example rule, you must associate it with your AppSync API. You can perform this step in the AWS CLI by running the following command.

```
aws waf associate-web-acl --web-acl-id waf-web-acl-arn --resource-arn appsync-api-arn
```

Event API runtime reference

The following sections contain the APPSYNC_JS runtime reference:

- Runtime reference overview — Learn more about how runtime data sources work in AWS AppSync Events.
- Context reference — Learn more about the context object and how it's used in AWS AppSync Events handlers.
- Runtime features — Learn more about supported runtime features.
- Runtime reference for DynamoDB — Learn more about how AWS AppSync Events handlers interact with DynamoDB.
- Runtime reference for OpenSearch Service — Learn more about how AWS AppSync Events handlers interact with Amazon OpenSearch Service.
- Runtime reference for Lambda — Learn more about how AWS AppSync Events API handlers interact with AWS Lambda.
- Runtime reference for EventBridge — Learn more about how AWS AppSync Events API handlers interact with Amazon EventBridge.
- Runtime reference for HTTP — Learn more about how AWS AppSync Events API handlers interact with HTTP endpoints.
- Runtime reference for Amazon RDS — Learn more about how AWS AppSync Events API handlers interact with Amazon Relational Database Service.
- Runtime reference for Amazon Bedrock — Learn more about how how AWS AppSync Events API handlers interact with Amazon Bedrock.

AWS AppSync Event API runtime reference

AWS AppSync lets you respond to specific triggers that occur in the service with code that runs on AWS AppSync's JavaScript runtime (APPSYNC_JS).

The AWS AppSync JavaScript runtime provides a subset of JavaScript libraries, utilities, and features. For a complete list of features and functionality supported by the APPSYNC_JS runtime, see Runtime features.

Topics

Event handlers overview

An event handler is a function defined in a namespace that is invoked by specific triggers in the system. Currently you can define onPublish and onSubscribe event handlers: handlers that respond to events being published to a channel in the namespace, and handlers that respond to subscription items on a channel in the namespace. An onPublish handler is called before events are broadcast to subscribed clients, giving you a chance to transform the events first. An onSubscribe handler is called as a client tries to subscribe, giving you the chance to accept or reject the subscription attempt.

Event handlers are optional and are not required for your channel namespaces to be effective.

Writing event handlers

A handler is defined by a single function that doesn't interact with a data source, or is defined by an object that implements a request and a response function to interact with one of your data sources. When working with a Lambda function data source, AWS AppSync Events supports a `DIRECT` integration that allows you to interact with your Lambda function without writing any handler code.

You provide your code for the handlers using the namespace's code property. Essentially, you use a "single file" to define all your handlers. In your code file, you identify your handler definitions by exporting a function or object named onPublish or onSubscribe.

Handler with no data source integration

You can define a handler with no data source integration. In this case, the handler is defined as a single function. In the following example, the onPublish handler shows the default behavior when no handler is defined. It simply forwards the list of events.

```
export function onPublish(ctx) {
  return ctx.events
}
```

As an alternate example, this definition returns an empty list, which means that no event will be broadcast to the subscribers.

```
export const onPublish = (ctx) => ([])
```

In this example, the handler only returns the list of published events, but adds the property `msg` to the payload.

```
export function onPublish(ctx) {
  return ctx.events.map(({id, payload}) => {
    return {id: payload: {...payload, msg: "Hello!"}}
  })
}
```

Handler with a data source integration

You can define a handler with a data source integration. In this case, you define an object that implements and `request` and a `response` function. The `request` function defines the payload that is sent to invoke the data source while the `response` function receives the result of that invocation. The list of events to broadcast is returned by the `response` function.

The following example defines an `onPublish` handler that saves all published events to a `messages_table` before forwarding them to be broadcast. The `onSubscribe` handler doesn't have a data source integration and is defined by a single function that simply logs a message.

```
import * as ddb from `@aws-appsync/utils/dynamodb`

const TABLE = 'messages_table'
export const onPublish = {
  request(ctx) {
    const channel = ctx.info.channel.path
    return ddb.batchPut({
      tables: {
        [TABLE]: ctx.events.map(({ id, payload }) => ({ channel, id, ...payload })),
      },
    })
  },
  response(ctx) {
    console.log(`Batch Put result:`, ctx.result.data[TABLE])
    return ctx.events
  },
```

```
}

export const onSubscribe = (ctx) => {
  console.debug(`Joined the chat: ${ctx.info.channel.path}`)
}
```

Skipping the data source

You might have situations where you need to skip the data source invocation at run time. You can do this by using the `runtime.earlyReturn` utility. `earlyReturn` immediately returns the provided payload and skips the response function.

```
import * as ddb from `@aws-appsync/utils/dynamodb`

const TABLE = 'messages_table'
export const onPublish = {
  request(ctx) {
    if (ctx.info.channel.segments.includes('private')) {
      // return early and do no execute the response.
      return runtime.earlyReturn(ctx.events)
    }
    const channel = ctx.info.channel.path
    return ddb.batchPut({
      tables: {
        [TABLE]: ctx.events.map(({ id, payload }) => ({ channel, id, ...payload })),
      },
    })
  },
  response(ctx) {
    return ctx.result.data[TABLE].map(({ id, ...payload }) => ({ id, payload }))
  },
}
```

Returning an error

During the execution of an event handler, you might need to return an error back to the publisher or subscriber. Use the `util.error` function to do this. When publishing is done using the HTTP endpoint, this returns an HTTP 403 response. When publishing over WebSocket, this returns a `publish_error` message with the provided message. The following example demonstrates how to return an error message.

```
export function onPublish(ctx) {
```

```
  util.error("Not possible!")
  return ctx.events
}
```

Unauthorizing a request

Your event handlers are always called after AWS AppSync has authorized the requests. However, you can run additional business logic and unauthorize a request in your event handler using the `util.unauthorize` function. When publishing over HTTP, this returns an HTTP 401 response. Over WebSocket, this returns a `publish_error` message with an `UnauthorizedException` error type. When trying to connect over WebSocket, you get a `subscribe_error` with an `Unauthorized` error type.

```
export function onSubscribe(ctx) {
  if (somethingNotValid() === true) {
    util.unauthorized()
  }
}

function somethingNotValid() {
  // implement custom business logic
}
```

Direct Lambda integration

AWS AppSync lets you integrate Lambda functions directly with your channel namespace without writing additional handler code. This integration supports both publish and subscribe operations through Request/Response mode.

How it works

When AWS AppSync calls your Lambda function, it passes a context object containing event information. Then, the Lambda function can perform the following operations:

- Filter and transform events for broadcasting
- Return error messages for failed processing
- Handle both publish and subscribe operations

Publish operation response format

For onPublish handlers, your Lambda function must return a response payload with the following structure:

```
type LambdaAppSyncEventResponse = {
  /** Array of outgoing events to broadcast */
  events?: OutgoingEvent[],

  /** Optional error message if processing fails */
  error?: string
}
```

> **Note**
>
> If you include an error message in the response, AWS AppSync logs it (when logging is enabled) but doesn't return it to the publisher.

Subscribe operation response

For onSubscribe handlers, your Lambda function must return one of the following:

- A payload containing an error message
- null to indicate a successful subscription

```
type LambdaAppSyncEventResponse = {
  /** Error message if subscription fails */
  error: string
} | null
```

Best practices

We recommend the following best practices for direct Lambda integrations:

- Enable logging to track error messages.
- Ensure your Lambda function handles both success and error cases.
- Test your integration with various payload scenarios.

Utilizing Powertools for Lambda

You can utilize Powertools for Lambda to efficiently write your Lambda function handlers. To learn more, see the following Powertools for AWS Lambda documentation resources:

- TypeScript/Node.js — See https://docs.powertools.aws.dev/lambda/typescript/latest/features/event-handler/appsync-events/ in the *Powertools for AWS Lambda (TypeScript)* documentation.
- Python — See https://docs.powertools.aws.dev/lambda/python/latest/core/event_handler/appsync_events/ in the *Powertools for AWS Lambda (Python)* documentation.
- .NET — See https://docs.powertools.aws.dev/lambda/dotnet/core/event_handler/appsync_events/ in the *Powertools for AWS Lambda (.NET)* documentation.

Configuring utilities for the APPSYNC_JS runtime

AWS AppSync provides the following two libraries that help you develop event handlers with the APPSYNC_JS runtime:

- `@aws-appsync/eslint-plugin` - Catches and fixes problems quickly during development.
- `@aws-appsync/utils` - Provides type validation and autocompletion in code editors.

Configuring the eslint plugin

ESLint is a tool that statically analyzes your code to quickly find problems. You can run ESLint as part of your continuous integration pipeline. `@aws-appsync/eslint-plugin` is an ESLint plugin that catches invalid syntax in your code when leveraging the APPSYNC_JS runtime. The plugin allows you to quickly get feedback about your code during development without having to push your changes to the cloud.

`@aws-appsync/eslint-plugin` provides two rule sets that you can use during development.

"plugin:@aws-appsync/base" configures a base set of rules that you can leverage in your project. The following table describes these rules.

Rule	Description
no-async	Async processes and promises are not supported.

Rule	Description
no-await	Async processes and promises are not supported.
no-classes	Classes are not supported.
no-for	`for` is not supported (except for `for-in` and `for-of`, which are supported)
no-continue	`continue` is not supported.
no-generators	Generators are not supported.
no-yield	`yield` is not supported.
no-labels	Labels are not supported.
no-this	`this` keyword is not supported.
no-try	Try/catch structure is not supported.
no-while	While loops are not supported.
no-disallowed-unary-operators	`++`, `--`, and ~ unary operators are not allowed.
no-disallowed-binary-operators	The `instanceof` operator is not allowed.
no-promise	Async processes and promises are not supported.

"plugin:@aws-appsync/recommended" provides some additional rules but also requires you to add TypeScript configurations to your project.

Rule	Description
no-recursion	Recursive function calls are not allowed.

Rule	Description
no-disallowed-methods	Some methods are not allowed. See the Runtime features for a full set of supported built-in functions.
no-function-passing	Passing functions as function arguments to functions is not allowed.
no-function-reassign	Functions cannot be reassigned.
no-function-return	Functions cannot be the return value of functions.

To add the plugin to your project, follow the installation and usage steps at Getting Started with ESLint. Then, install the plugin in your project using your project package manager (e.g., npm, yarn, or pnpm):

```
$ npm install @aws-appsync/eslint-plugin
```

In your `.eslintrc.{js,yml,json}` file, add **"plugin:@aws-appsync/base"** or **"plugin:@aws-appsync/recommended"** to the `extends` property. The snippet below is a basic sample `.eslintrc` configuration for JavaScript:

```
{
  "extends": ["plugin:@aws-appsync/base"]
}
```

To use the **"plugin:@aws-appsync/recommended"** rule set, install the required dependency:

```
$ npm install -D @typescript-eslint/parser
```

Then, create an `.eslintrc.js` file:

```
{
  "parser": "@typescript-eslint/parser",
  "parserOptions": {
    "ecmaVersion": 2018,
```

```
    "project": "./tsconfig.json"
  },
  "extends": ["plugin:@aws-appsync/recommended"]
}
```

Bundling, TypeScript, and source maps for the APPSYNC_JS runtime

TypeScript enhances AWS AppSync development by providing type safety and early error detection. You can write TypeScript code locally and transpile it to JavaScript before using it with the APPSYNC_JS runtime. The process starts with installing TypeScript and configuring tsconfig.json for the APPSYNC_JS environment. You can then use bundling tools like esbuild to compile and bundle the code.

You can leverage custom and external libraries in your handler and function code, as long as they comply with APPSYNC_JS requirements. Bundling tools combine code into a single file for use in AWS AppSync. Source maps can be included to aid debugging.

Leveraging libraries and bundling your code

In your handler code, you can leverage both custom and external libraries so long as they comply with the APPSYNC_JS requirements. This makes it possible to reuse existing code in your application. To make use of libraries that are defined by multiple files, you must use a bundling tool, such as esbuild, to combine your code in a single file that can then be saved to your AWS AppSync namespace handler code.

When bundling your code, keep the following in mind:

- APPSYNC_JS only supports ECMAScript modules (ESM).
- `@aws-appsync/*` modules are integrated into APPSYNC_JS and should not be bundled with your code.
- The APPSYNC_JS runtime environment is similar to NodeJS in that code does not run in a browser environment.
- You can include an optional source map. However, do not include the source content.

  To learn more about source maps, see Using source maps.

For example, to bundle your handler code located at `src/appsync/onPublish.js`, you can use the following esbuild CLI command:

```
$ esbuild --bundle \
--sourcemap=inline \
--sources-content=false \
--target=esnext \
--platform=node \
--format=esm \
--external:@aws-appsync/utils \
--outdir=out/appsync \
 src/appsync/onPublish.js
```

Building your code and working with TypeScript

TypeScript is a programming language developed by Microsoft that offers all of JavaScript's features along with the TypeScript typing system. You can use TypeScript to write type-safe code and catch errors and bugs at build time before saving your code to AWS AppSync. The `@aws-appsync/utils` package is fully typed.

The APPSYNC_JS runtime doesn't support TypeScript directly. You must first transpile your TypeScript code to JavaScript code that the APPSYNC_JS runtime supports before saving your code to AWS AppSync. You can use TypeScript to write your code in your local integrated development environment (IDE), but note that you cannot create TypeScript code in the AWS AppSync console.

To get started, make sure you have TypeScript installed in your project. Then, configure your TypeScript transcompilation settings to work with the APPSYNC_JS runtime using TSConfig. Here's an example of a basic `tsconfig.json` file that you can use:

```
// tsconfig.json
{
  "compilerOptions": {
    "target": "esnext",
    "module": "esnext",
   "noEmit": true,
   "moduleResolution": "node",
  }
}
```

You can then use a bundling tool like esbuild to compile and bundle your code. For example, given a project with your AWS AppSync code located at `src/appsync`, you can use the following command to compile and bundle your code:

```
$ esbuild --bundle \
--sourcemap=inline \
--sources-content=false \
--target=esnext \
--platform=node \
--format=esm \
--external:@aws-appsync/utils \
--outdir=out/appsync \
 src/appsync/**/*.ts
```

Using generics in TypeScript

You can use generics with several of the provided types. For example, you can write a handler that makes use of the √≈. In your IDE, type definitions and auto-complete hints will guide you into properly using the available utilities.

```
import type { EventOnPublishContext, IncomingEvent, OutgoingEvent } from "@aws-appsync/
utils"
import * as ddb from '@aws-appsync/utils/dynamodb'

type Message = {
  id: string;
  text: string;
  owner: string;
  likes: number
}

type OnP<T = any> = {
  request: (ctx: EventOnPublishContext<T>) => unknown,
  response: (ctx: EventOnPublishContext<T>) => OutgoingEvent[] | IncomingEvent[]
}

export const onPublish: OnP<Message> = {
  request(ctx) {
    const msg = ctx.events[0]
    return ddb.update<Message>({
      key: { owner: msg.payload.owner, id: msg.payload.id },
      update: msg.payload,
      condition: { id: { attributeExists: true } }
    })
  },
  response: (ctx) => ctx.events
```

```
}
```

Linting your bundles

You can automatically lint your bundles by importing the `esbuild-plugin-eslint` plugin. You can then enable it by providing a `plugins` value that enables eslint capabilities. Below is a snippet that uses the esbuild JavaScript API in a file called `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 })],
})
```

Using source maps

You can provide an inline source map (`sourcemap`) with your JavaScript code. Source maps are useful for when you bundle JavaScript or TypeScript code and want to see references to your input source files in your logs and runtime JavaScript error messages.

Your `sourcemap` must appear at the end of your code. It is defined by a single comment line that follows the following format:

```
//# sourceMappingURL=data:application/json;base64,<base64 encoded string>
```

The following is an example of a source map:

```
//# sourceMappingURL=data:application/
json;base64,ewogICJ2ZXJzaW9uIjogMywKICAic291cmNlcyI6IFsibGliLmpzIiwgImNvZGUuanMiXSwKICAibWFwcGl
```

Source maps can be created with esbuild. The example below shows you how to use the esbuild JavaScript API to include an inline source map when code is built and bundled:

```
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 the preceeding example, the `sourcemap` and `sourcesContent` options specify that a source map should be added in line at the end of each build but should not include the source content. As a convention, we recommend not including source content in your `sourcemap`. You can disable this in esbuild by setting `sources-content` to `false`.

To illustrate how source maps work, review the following example in which handler code references helper functions from a helper library. The code contains log statements in the handler code and in the helper library:

./src/channelhandler.ts (your handler)

```
import { EventOnPublishContext }  from "@aws-appsync/utils";
 import { mapper }  from "./lib/mapper";

 exportfunction onPublish ( ctx: EventOnPublishContext ) {
   return ctx.events.map(mapper)
}
```

./lib/helper.ts (a helper file)

```
import { IncomingEvent, OutgoingEvent } from "@aws-appsync/utils";

export function mapper(event: IncomingEvent, index: number) {
  console.log(`-> mapping: event ${event.id}`)
  return {
    ...event,
    payload: { ...event.payload, mapped: true },
    error: index % 2 === 0 ? 'flip flop error' : null
  } as OutgoingEvent
}
```

When you build and bundle the handler file, your handler code will include an inline source map. When your handler runs, entries will appear in the CloudWatch logs.

AWS AppSync Event API context reference

AWS AppSync defines a set of variables and functions for working with handlers. This topic describes these functions and provides examples.

Accessing the `context`

The `context` argument of a request and response handler is an object that holds all of the contextual information for your handler invocation. It has the following structure:

```
type Context = {
  identity?: Identity;
  result?: any;
  request: Request;
  info: EventsInfo;
  stash: any;
  error?: Error
  events?: IncomingEvent[];
}
```

> **Note**
>
> The `context` object is commonly referred to as `ctx`.

Each field in the `context` object is defined as follows:

`context` fields

`identity`

An object that contains information about the caller. For more information about the structure of this field, see Identity.

`result`

A container for the results of this handler when a data source is configured, available in the response function of a namespace handler.

`request`

A container for the headers and information about the custom domain that was used.

`info`

An object that contains information about the operation on the channel namespace. For the structure of this field, see Info.

`stash`

The stash is an object that is made available inside each handler. The same stash object lives through a single handler evaluation. You can use the stash to pass arbitrary data across request and response functions of your handlers.

You can add items to the stash as follows:

```
ctx.stash.newItem = { key: "something" }
Object.assign(ctx.stash, {key1: value1, key2: value})
```

You can remove items from the stash by modifying the following code:

```
delete ctx.stash.key
```

Identity

The `identity` section contains information about the caller. The shape of this section depends on the authorization type of your AWS AppSync API. For more information about security options, see Configuring authorization and authentication to secure Event APIs.

`API_KEY` authorization

The `identity` field isn't populated.

`AWS_LAMBDA` authorization

The `identity` has the following form:

```
type AppSyncIdentityLambda = {
  handlerContext: any;
};
```

The `identity` contains the `handlerContext` key, containing the same `handlerContext` content returned by the Lambda function authorizing the request.

`AWS_IAM` authorization

The `identity` has the following form:

```
type AppSyncIdentityIAM = {
  accountId: string;
  cognitoIdentityPoolId: string;
  cognitoIdentityId: string;
  sourceIp: string[];
  username: string;
  userArn: string;
  cognitoIdentityAuthType: string;
  cognitoIdentityAuthProvider: string;
};
```

`AMAZON_COGNITO_USER_POOLS` authorization

The `identity` has the following form:

```
type AppSyncIdentityCognito = {
  sourceIp: string[];
  username: string;
  groups: string[] | null;
  sub: string;
  issuer: string;
  claims: any;
  defaultAuthStrategy: string;
};
```

Each field is defined as follows:

`accountId`

The AWS account ID of the caller.

`claims`

The claims that the user has.

`cognitoIdentityAuthType`

Either authenticated or unauthenticated based on the identity type.

`cognitoIdentityAuthProvider`

A comma-separated list of external identity provider information used in obtaining the credentials used to sign the request.

`cognitoIdentityId`

The Amazon Cognito identity ID of the caller.

`cognitoIdentityPoolId`

The Amazon Cognito identity pool ID associated with the caller.

`defaultAuthStrategy`

The default authorization strategy for this caller (`ALLOW` or `DENY`).

`issuer`

The token issuer.

`sourceIp`

The source IP address of the caller that AWS AppSync receives. If the request doesn't include the `x-forwarded-for` header, the source IP value contains only a single IP address from the TCP connection. If the request includes a `x-forwarded-for` header, the source IP is a list of IP addresses from the `x-forwarded-for` header, in addition to the IP address from the TCP connection.

`sub`

The UUID of the authenticated user.

`user`

The IAM user.

`userArn`

The Amazon Resource Name (ARN) of the IAM user.

`username`

The user name of the authenticated user. In the case of `AMAZON_COGNITO_USER_POOLS` authorization, the value of *username* is the value of attribute *cognito:username*. In the case of `AWS_IAM` authorization, the value of *username* is the value of the AWS user principal. If you're using IAM authorization with credentials vended from Amazon Cognito identity pools, we recommend that you use `cognitoIdentityId`.

Request property

The `request` property contains the headers that were sent with the request, and the custom domain name if it was used.

Request headers

The headers sent in HTTP requests to your API.

AWS AppSync supports passing custom headers from clients and accessing them in your handlers by using `ctx.request.headers`. You can then use the header values for actions such as inserting data into a data source or authorization checks. You can use single or multiple request headers.

If you set a header of `animal` with a value of `duck` as in the following example:

```
curl --location "https://YOUR_EVENT_API_ENDPOINT/event" \
--header 'Content-Type: application/json' \
--header "x-api-key:ABC123" \
--header "animal:duck" \
--data '{ "channel":"/news", "events":["\"Breaking news!\""] }'
```

Then, this could then be accessed with `ctx.request.headers.animal`.

You can also pass multiple headers in a single request and access these in the handler. For example, if the `custom` header is set with two values as follows:

```
curl --location "https://YOUR_EVENT_API_ENDPOINT/event" \
--header 'Content-Type: application/json' \
--header "x-api-key:ABC123" \
--header "animal:duck" \
--header "animal:goose" \
--data '{ "channel":"/news", "events":["\"Breaking news!\""] }'
```

You could then access these as an array, such as `ctx.request.headers.animal[1]`.

> **Note**
>
> AWS AppSync doesn't expose the cookie header in `ctx.request.headers`.

Access the request custom domain name

AWS AppSync supports configuring a custom domain that you can use to access your HTTP and WebSocket real-time endpoints for your APIs. When making a request with a custom domain name, you can get the domain name using `ctx.request.domainName`. When using the default endpoint domain name, the value is `null`.

Info property

The `info` section contains information about the request made to your channel namespace. This section has the following form:

```
type EventsInfo = {
  info: {
    channel: {
      path: string;
      segments: string[];
    }
  };
  channelNamespace: {
    name: string
  }
  operation: 'SUBSCRIBE' | 'PUBLISH'
}
```

Each field is defined as follows:

`info.channel.path`

The channel path the operation is executed on, for example, `/default/user/johm`.

`info.channel.segments`

The segments of the channel path, for example, `['default', 'user', 'john']`.

`info.channelNamespace.name`

The name of the channel namespace, for example, `'default'`.

`info.operation`

The operation executed: `PUBLISH` or `SUBSCRIBE`.

Runtime features

The APPSYNC_JS runtime environment provides features and utilities to help you work with data, and write functions and AWS AppSync Event API handlers. The topics in this section describe the language features that are supported for AWS AppSync Event APIs.

Topics

Supported runtime features

The APPSYNC_JS runtime supports the features described in the following sections.

Topics

Core features

The following core features are supported.

Types

The following types are supported:

- numbers
- strings
- booleans
- objects
- arrays
- functions

Operators

The following operators are supported:

- standard math operators (`+`, `-`, `/`, `%`, `*`, etc.)
- nullish coalescing operator (`??`)
- Optional chaining (`?.`)
- bitwise operators
- `void` and `typeof` operators
- spread operators (`...`)

The following operators are not supported:

- unary operators (`++`, `--`, and `~`)
- `in` operator

> **Note**
>
> Use the `Object.hasOwn` operator to check if the specified property is in the specified object.

Statements

The following statements are supported:

- `const`
- `let`
- `var`
- `break`
- `else`
- `for-in`
- `for-of`
- `if`
- `return`
- `switch`
- spread syntax

The following are not supported:

- `catch`
- `continue`
- `do-while`
- `finally`
- `for(initialization; condition; afterthought)`

> **Note**
>
> The exceptions are `for-in` and `for-of` expressions, which are supported.

- `throw`
- `try`
- `while`
- labeled statements

Literals

The following ES 6 template literals are supported:

- Multi-line strings
- Expression interpolation
- Nesting templates

Functions

The following function syntax is supported:

- Function declarations are supported.
- ES 6 arrow functions are supported.
- ES 6 rest parameter syntax is supported.

Primitive objects

The following primitive objects of ES and their functions are supported.

Object

The following objects are supported:

- `Object.assign()`
- `Object.entries()`
- `Object.hasOwn()`
- `Object.keys()`
- `Object.values()`
- `delete`

String

The following strings are supported:

- `String.prototype.length()`
- `String.prototype.charAt()`
- `String.prototype.concat()`
- `String.prototype.endsWith()`
- `String.prototype.indexOf()`
- `String.prototype.lastIndexOf()`
- `String.raw()`
- `String.prototype.replace()`

> **Note**
>
> Regular expressions are not supported.

However, Java-styled regular expression constructs are supported in the provided parameter. For more information see Pattern.

- `String.prototype.replaceAll()`

> **Note**
>
> Regular expressions are not supported.
> However, Java-styled regular expression constructs are supported in the provided parameter. For more information see 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

The following numbers are supported:

- `Number.isFinite`
- `Number.isNaN`

Built-in objects and functions

The following functions and objects are supported.

Math

- `Math.random()`
- `Math.min()`
- `Math.max()`
- `Math.round()`

- `Math.floor()`
- `Math.ceil()`

Array

- `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()` doesn't support arguments.

- `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

The console object is available for debugging. During live query execution, console log/error statements are sent to Amazon CloudWatch Logs (if logging is enabled). During code evaluation with `evaluateCode`, log statements are returned in the command response.

- `console.error()`
- `console.log()`

Function

- The `apply`, `bind`, and `call` methods not are supported.
- Function constructors are not supported.
- Passing a function as an argument is not supported.
- Recursive function calls are not supported.

JSON

The following JSON methods are supported:

- `JSON.parse()`

> **Note**
>
> Returns a blank string if the parsed string is not valid JSON.

- `JSON.stringify()`

Promises

Async processes are not supported, and promises are not supported.

> **Note**
>
> Network and file system access is not supported within the `APPSYNC_JS` runtime in AWS AppSync. AWS AppSync handles all I/O operations based on the requests made by the AWS AppSync handler or AWS AppSync function.

Globals

The following global constants are supported:

- `NaN`
- `Infinity`
- `undefined`
- `util`
- `extensions`
- `runtimeb`

Error types

Throwing errors with `throw` is not supported. You can return an error by using `util.error()` function. You can include an error in your handler response by using the `util.appendError` function.

Built-in utilities

The `util` variable contains general utility methods to help you work with data. Unless otherwise specified, all utilities use the UTF-8 character set.

Encoding utils

`util.urlEncode(String)`

Returns the input string as an `application/x-www-form-urlencoded` encoded string.

`util.urlDecode(String)`

Decodes an `application/x-www-form-urlencoded` encoded string back to its non-encoded form.

`util.base64Encode(string) : string`

Encodes the input into a base64-encoded string.

`util.base64Decode(string) : string`

Decodes the data from a base64-encoded string.

Built-in modules

Modules are a part of the APPSYNC_JS runtime and provide utilities to help write functions and Event API handlers. This section describes the DynamoDB and Amazon RDS module functions that you can use to interact with these data sources.

Amazon DynamoDB built-in module

The DynamoDB module functions provide an enhanced experience when interacting with DynamoDB data sources. You can make requests toward your DynamoDB data sources using the functions and without adding type mapping.

Modules are imported using `@aws-appsync/utils/dynamodb`:

```
import * as ddb from '@aws-appsync/utils/dynamodb';
```

DynamoDB `get()` function

The DynamoDB `get()` function generates a `DynamoDBGetItemRequest` object to make a `GetItem` request to DynamoDB.

Definition

```
get<T>(payload: GetInput): DynamoDBGetItemRequest
```

Example

The following example fetches an item from DynamoDB in a `subscribe` handler.

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

export const onSubscribe = {
  request(ctx) {
    return ddb.get({key: {
      path: ctx.info.channel.path,
      sub: ctx.identity.sub
    }})
  },
  response(ctx) {
    console.log('Got the item:', ctx.result)
    if (!ctx.result){
      console.error("No info about this user for this channel path.")
```

```
      until.unauthorized()
    }
  }
}
```

DynamoDB `query()` function

The DynamoDB `query()` function generates a `DynamoDBQueryRequest` object to make a `Query` request to DynamoDB.

Definition

```
query<T>(payload: QueryInput): DynamoDBQueryRequest
```

Example

The following example performs a query against a DynamoDB table.

```
import * as ddb from '@aws-appsync/utils/dynamodb'

export const onPublish = {
  request(ctx) {
    // Find all items from this channel that exist on this path
    return ddb.query<{ channel: string; path: string }>({
      query: {
        channel: { eq: ctx.info.channelNamespace.name },
        path: { beginsWith: ctx.info.channe.path },
      },
      projection: ['channel', 'path', 'msgId'],
    })
  },
  response(ctx) {
    // Broadcast items that have not been saved to the table
    const ids = ctx.result.items.map(({ msgId }) =>  msgId )
    return ctx.events.filter(({ payload: { msgId } }) => !ids.includes(msgId))
  },
}
```

DynamoDB `scan()` function

The DynamoDB `scan()` function generates a `DynamoDBScanRequest` object to make a `Scan` request to DynamoDB.

Definition

```
scan<T>(payload: ScanInput): DynamoDBScanRequest
```

Example

The following example scans all items in a DynamoDB table.

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export const onPublish = {
  request(ctx){
    return ddb.scan({
      limit: 20,
      projection: ['channel', 'path', 'msgId'],
      filter: { status: { eq: 'ACTIVE' } }
    })
  },
  response: (ctx) => ctx.events
}
```

DynamoDB `put()` function

The DynamoDB `put()` function generates a `DynamoDBPutItemRequest` object to make a `PutItem` request to DynamoDB.

Definition

```
put<T>(payload: PutInput): DynamoDBPutItemRequest
```

Example

The following example saves an event to a DynamoDB table in an `OnPublish` handler.

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export const onPublish = {
  request(ctx) {
    const {id, payload: item} = ctx.events[0]
    return ddb.put({ key: {id}, item })
  },
  response: (ctx) => ctx.events
```

```
}
```

DynamoDB `remove()` function

The DynamoDB `remove()` function generates a `DynamoDBDeleteItemRequest` object to make a `DeleteItem` request to DynamoDB.

Definition

```
remove<T>(payload: RemoveInput): DynamoDBDeleteItemRequest
```

Example

This `OnPublish` handler deletes an item in a DynamoDB table and forwards an empty list. No event is broadcast.

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export const onPublish = {
  request(ctx) {
    const { id } = ctx.events[0]
    return ddb.remove({key: id});
  },
  response: (ctx) => ([])
}
```

DynamoDB `update()` function

The DynamoDB `update()` function generates a `DynamoDBUpdateItemRequest` object to make an `UpdateItem` request to DynamoDB.

Definition

```
update<T>(payload: UpdateInput): DynamoDBUpdateItemRequest
```

Example

This `OnPublish` handler increases the account received item before it is broadcast.

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export const onPublish = {
```

```
  request(ctx) {
    const { id, payload } = ctx.events[0]
    return ddb.update({
      key: { id },
      condition: { version: { eq: payload.version } },
      update: { ...payload, version: ddb.operations.increment(1) },
    });
  },
  response: (ctx) => ctx.events
}
```

DynamoDB `batchGet()` function

The DynamoDB `batchGet()` function generates a `DynamoDBBatchGetItemRequest` object to make an `BatchGetItem` request to retrieve multiple items from one or more DynamoDB tables.

Definition

```
batchGet<T>(payload: BatchGetInput): DynamoDBBatchGetItemRequest
```

Example

The following example retrieves multiple items from a DynamoDB table in a single request/

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export const onPublish = {
  request(ctx) {
    return ddb.batchGet({
      tables: {
        users: {
          keys: ctx.events.map(e => ({id: e.payload.id})),
          projection: ['id', 'name', 'email']
        }
      }
    })
  },
  response(ctx) {
    const users = ctx.result.data.users.reduce((acc, cur) => {
      acc[cur.id] = cur
    }, {})
    return ctx.events.map(event => {
      return {
```

```
        id: event.id,
        payload: {...event.payload, ...users[event.payload.id]}
      }
    })
  }
}
```

DynamoDB `batchPut()` function

The DynamoDB `batchput()` function generates a `DynamoDBBatchPutItemRequest` object to make an `BatchWriteItem` request to put multiple items into one or more DynamoDB tables.

Definition

```
batchPut<T>(payload: BatchPutInput): DynamoDBBatchPutItemRequest
```

Example

The following example writes multiple items to a DynamoDB table in a single request.

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export const onPublish = {
  request(ctx) {
    return ddb.batchPut({
      tables: {
        messages: ctx.events.map(({ id, payload }) => ({
          channel: ctx.info.channelNamespace.name,
          id,
          ...payload
        })),
      }
    });
  },
  response: (ctx) => ctx.events
}
```

DynamoDB `batchDelete()` function

The DynamoDB `batchDelete()` function generates a `DynamoDBBatchDeleteItemRequest` object to make an `BatchWriteItem` request to delete multiple items from one or more DynamoDB tables.

Definition

```
batchDelete(payload: BatchDeleteInput): DynamoDBBatchDeleteItemRequest
```

Example

The following example deletes multiple items from a DynamoDB table in a single request.

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export const onPublish = {
  request(ctx) {
    const name = ctx.info.channelNamespace.name
    return ddb.batchDelete({
      tables: {
        [name]: ctx.events.map(({ payload }) => ({ id: payload.id })),
      }
    });
  },
  response: (ctx) => ([])
}
```

DynamoDB `transactGet()` function

The DynamoDB `transactGet()` function generates a `DynamoDBTransactGetItemsRequest` object to make an `TransactGetItems` request to retrieve multiple items with strong consistency in a single atomic transaction.

Definition

```
transactGet(payload: TransactGetInput): DynamoDBTransactGetItemsRequest
```

Example

The following example retrieves multiple items in a single atomic transaction.

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export const onPublish = {
  request(ctx) {
    return ddb.transactGet({
      items: ctx.events.map(event => ({
```

```
        table: event.payload.table,
        key: { id: event.payload.id },
        projection: [...event.payload.fields]
      }))
    })
  },
  response(ctx) {
    items = ctx.result.items
    return ctx.events.map((event, i) => ({
      id: event.id,
      payload: { ...event.payload, ...items[i] }
    }))
  }
}
```

DynamoDB `transactWrite()` function

The DynamoDB `transactWrite()` function generates a `DynamoDBTransactWriteItemsRequest` object to make an `TransactWriteItems` request to perform multiple write operations in a single atomic transaction.

Definition

```
transactWrite(payload: TransactWriteInput): DynamoDBTransactWriteItemsRequest
```

Example

The following example performs multiple write operations in a single atomic transaction.

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export const onPublish = {
  request(ctx) {
    const order = ctx.events[0]
    return ddb.transactWrite({
      items: [
        {
          putItem: {
            table: 'Orders',
            key: { id: order.payload.id },
            item: {
              status: 'PENDING',
              createdAt: util.time.toISOString(),
```

```
              items: order.items.map(({ id }) => id)
            }
          }
        },
        ...(order.items.map(({ id, item }) => ({
          putItem: {
            table: 'Items',
            key: { orderId: order.payload.id, id },
            item
          }
        })))
      ]
    });
  },
  response: (ctx) => ctx.events
}
```

DynamoDB set utilities

The `@aws-appsync/utils/dynamodb` provides the following `set` utility functions that you can use to work with string sets, number sets, and binary sets.

`toStringSet`

Converts a list of strings to the DynamoDB string set format.

`toNumberSet`

Converts a list of numbers to the DynamoDB string set format.

`toBinarySet`

Converts a list of binary to the DynamoDB string set format.

Example

The following example converts a list of strings to DynamoDB string set format.

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export const onPublish = {
  request(ctx) {
    const { id, payload } = ctx.events[0]
    return ddb.update({
```

```
      key: { id },
      update: {segments: ddb.toStringSet(ctx.info.channel.segments)},
    });
  },
  response: (ctx) => ctx.events
}
```

DynamoDB conditions and filters

You can use the following operators to create filters and conditions.

Operator	Description	Possible value types
eq	Equal	number, string, boolean
ne	Not equal	number, string, boolean
le	Less than or equal	number, string
lt	Less than	number, string
ge	Greater than or equal	number, string
gt	Greater than	number, string
contains	Like	string
notContains	Not like	string
beginsWith	Starts with prefix	string
between	Between two values	number, string
attributeExists	The attribute is not null	number, string, boolean
size	checks the length of the element	string

You can combine these operators with `and`, `or`, and `not`.

```
const condition = {
```

```
  and: [
    { name: { eq: 'John Doe' }},
    { age: { between: [10, 30] }},
    {or: [
      {id :{ attributeExists: true}}
    ]}
  ]
}
```

DynamoDB operations

The DynamoDB operations object provides utility functions for common DynamoDB operations. These utilities are particularly useful in update() function calls.

The following operations are available:

`add(value)`

A helper function that adds a a value to the item when updating DynamoDB.

`remove()`

A helper function that removes an attribute from an item when updating DynamoDB.

`replace(value)`

A helper function that replaces an existing attribute when updating an item in DynamoDB. This is useful for when you want to update the entire object or sub-object in the attribute.

`increment(amount)`

A helper function that increments a numeric attribute by the specified amount when updating DynamoDB.

`decrement(amount)`

A helper function that decrements a numeric attribute by the specified amount when updating DynamoDB.

`append(value)`

A helper function that appends a value to a list attribute in DynamoDB.

`prepend(value)`

A helper function that prepends a value to a list attribute in DynamoDB.

`updateListItem(value, index)`

A helper function that updates an item in a list.

Example

The following example demonstrates how to use various operations in an update request.

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export const onPublish = {
request(ctx) {
  return ddb.update({
    key: { id },
    update: {
      counter: ddb.operations.increment(1),
      tags: ddb.operations.append(['things']),
      items: ddb.operations.add({key: 'value'}),
      oldField: ddb.operations.remove(),
    },
  });
}

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

Inputs

`Type GetInput<T>`

```
GetInput<T>: {
    consistentRead?: boolean;
    key: DynamoDBKey<T>;
}
```

Type Declaration

- `consistentRead?: boolean` (optional)

  An optional boolean to specify whether you want to perform a strongly consistent read with DynamoDB.

- `key: DynamoDBKey<T>` (required)

  A required parameter that specifies the key of the item in DynamoDB. DynamoDB items may have a single hash key or hash and sort keys.

`Type PutInput<T>`

```
PutInput<T>: {
    _version?: number;
    condition?: DynamoDBFilterObject<T> | null;
    customPartitionKey?: string;
    item: Partial<T>;
    key: DynamoDBKey<T>;
    populateIndexFields?: boolean;
}
```

Type Declaration

- `_version?: number` (optional)
- `condition?: DynamoDBFilterObject<T> | null` (optional)

  When you put an object in a DynamoDB table, you can optionally specify a conditional expression that controls whether the request should succeed or not based on the state of the object already in DynamoDB before the operation is performed.

- `customPartitionKey?: string` (optional)

  When enabled, this string value modifies the format of the `ds_sk` and `ds_pk` records used by the delta sync table when versioning has been enabled. When enabled, the processing of the `populateIndexFields` entry is also enabled.

- `item: Partial<T>` (required)

  The rest of the attributes of the item to be placed into DynamoDB.

- `key: DynamoDBKey<T>` (required)

  A required parameter that specifies the key of the item in DynamoDB on which the put will be performed. DynamoDB items may have a single hash key or hash and sort keys.

- `populateIndexFields?: boolean` (optional)

  A boolean value that, when enabled along with the `customPartitionKey`, creates new entries for each record in the delta sync table, specifically in the `gsi_ds_pk` and `gsi_ds_sk`

columns. For more information, see Conflict detection and sync in the *AWS AppSync GraphQL Developer Guide*.

Type QueryInput<T>

```
QueryInput<T>: ScanInput<T> & {
    query: DynamoDBKeyCondition<Required<T>>;
}
```

Type Declaration

- `query: DynamoDBKeyCondition<Required<T>>` (required)

  Specifies a key condition that describes items to query. For a given index, the condition for a partition key should be an equality and the sort key a comparison or a `beginsWith` (when it's a string). Only number and string types are supported for partition and sort keys.

  Example

  Take the `User` type below:

  ```
  type User = {
    id: string;
    name: string;
    age: number;
    isVerified: boolean;
    friendsIds: string[]
  }
  ```

  The query can only include the following fields: `id`, `name`, and `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;
}
```

Type Declaration

- `_version?: number` (optional)
- `condition?: DynamoDBFilterObject<T>` (optional)

  When you remove an object in DynamoDB, you can optionally specify a conditional expression that controls whether the request should succeed or not based on the state of the object already in DynamoDB before the operation is performed.

  Example

  The following example is a `DeleteItem` expression containing a condition that allows the operation succeed only if the owner of the document matches the user making the request.

```
type Task = {
  id: string;
  title: string;
  description: string;
  owner: string;
  isComplete: boolean;
}
const condition: DynamoDBFilterObject<Task> = {
  owner: { eq: 'XXXXXXXXXXXXXXXX' },
}

remove<Task>({
   key: {
     id: 'XXXXXXXXXXXXXXXX',
  },
  condition,
});
```

- `customPartitionKey?: string` (optional)

  When enabled, the `customPartitionKey` value modifies the format of the `ds_sk` and `ds_pk` records used by the delta sync table when versioning has been enabled. When enabled, the processing of the `populateIndexFields` entry is also enabled.

- `key: DynamoDBKey<T>` (required)

A required parameter that specifies the key of the item in DynamoDB that is being removed. DynamoDB items may have a single hash key or hash and sort keys.

Example

If a `User` only has the hash key with a user `id`, then the key would look like this:

```
type User = {
 id: number
 name: string
 age: number
 isVerified: boolean
}
const key: DynamoDBKey<User> = {
 id: 1,
}
```

If the table user has a hash key (`id`) and sort key (`name`), then the key would look like this:

```
type User = {
 id: number
 name: string
 age: number
 isVerified: boolean
 friendsIds: string[]
}

const key: DynamoDBKey<User> = {
 id: 1,
 name: 'XXXXXXXXXX',
}
```

- `populateIndexFields?: boolean` (optional)

  A boolean value that, when enabled along with the `customPartitionKey`, creates new entries for each record in the delta sync table, specifically in the `gsi_ds_pk` and `gsi_ds_sk` columns.

`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;
}
```

Type Declaration

- `consistentRead?: boolean | null` (optional)

  An optional boolean to indicate consistent reads when querying DynamoDB. The default value is `false`.

- `filter?: DynamoDBFilterObject<T> | null` (optional)

  An optional filter to apply to the results after retrieving it from the table.

- `index?: string | null` (optional)

  An optional name of the index to scan.

- `limit?: number | null` (optional)

  An optional max number of results to return.

- `nextToken?: string | null` (optional)

  An optional pagination token to continue a previous query. This would have been obtained from a previous query.

- `scanIndexForward?: boolean | null` (optional)

  An optional boolean to indicate whether the query is performed in ascending or descending order. By default, this value is set to `true`.

- `segment?: number` (optional)

- `select?: DynamoDBSelectAttributes` (optional)

  Attributes to return from DynamoDB. By default, the AWS AppSync DynamoDB resolver only returns attributes that are projected into the index. The supported values are:

  - `ALL_ATTRIBUTES`

Returns all the item attributes from the specified table or index. If you query a local secondary index, DynamoDB fetches the entire item from the parent table for each matching item in the index. If the index is configured to project all item attributes, all of the data can be obtained from the local secondary index and no fetching is required.

- `ALL_PROJECTED_ATTRIBUTES`

  Returns all attributes that have been projected into the index. If the index is configured to project all attributes, this return value is equivalent to specifying `ALL_ATTRIBUTES`.

- `SPECIFIC_ATTRIBUTES`

  Returns only the attributes listed in `ProjectionExpression`. This return value is equivalent to specifying `ProjectionExpression` without specifying any value for `AttributesToGet`.

- `totalSegments?: number` (optional)

Type `DynamoDBSyncInput<T>`

```
DynamoDBSyncInput<T>: {
    basePartitionKey?: string;
    deltaIndexName?: string;
    filter?: DynamoDBFilterObject<T> | null;
    lastSync?: number;
    limit?: number | null;
    nextToken?: string | null;
}
```

Type Declaration

- `basePartitionKey?: string` (optional)

  The partition key of the base table to be used when performing a Sync operation. This field allows a Sync operation to be performed when the table utilizes a custom partition key.

- `deltaIndexName?: string` (optional)

  The index used for the Sync operation. This index is required to enable a Sync operation on the whole delta store table when the table uses a custom partition key. The Sync operation will be performed on the GSI (created on `gsi_ds_pk` and `gsi_ds_sk`).

- `filter?: DynamoDBFilterObject<T> | null` (optional)

An optional filter to apply to the results after retrieving it from the table.

- `lastSync?: number` (optional)

  The moment, in epoch milliseconds, at which the last successful Sync operation started. If specified, only items that have changed after `lastSync` are returned. This field should only be populated after retrieving all pages from an initial Sync operation. If omitted, results from the base table will be returned. Otherwise, results from the delta table will be returned.

- `limit?: number | null` (optional)

  An optional maximum number of items to evaluate at a single time. If omitted, the default limit will be set to `100` items. The maximum value for this field is `1000` items.

- `nextToken?: string | null` (optional)

Type DynamoDBUpdateInput<T>

```
DynamoDBUpdateInput<T>: {
    _version?: number;
    condition?: DynamoDBFilterObject<T>;
    customPartitionKey?: string;
    key: DynamoDBKey<T>;
    populateIndexFields?: boolean;
    update: DynamoDBUpdateObject<T>;
}
```

Type Declaration

- `_version?: number` (optional)
- `condition?: DynamoDBFilterObject<T>` (optional)

  When you update an object in DynamoDB, you can optionally specify a conditional expression that controls whether the request should succeed or not based on the state of the object already in DynamoDB before the operation is performed.

- `customPartitionKey?: string` (optional)

  When enabled, the `customPartitionKey` value modifies the format of the `ds_sk` and `ds_pk` records used by the delta sync table when versioning has been enabled. When enabled, the processing of the `populateIndexFields` entry is also enabled.

- `key: DynamoDBKey<T>` (required)

A required parameter that specifies the key of the item in DynamoDB that is being updated. DynamoDB items may have a single hash key or hash and sort keys.

- `populateIndexFields?: boolean` (optional)

  A boolean value that, when enabled along with the `customPartitionKey`, creates new entries for each record in the delta sync table, specifically in the `gsi_ds_pk` and `gsi_ds_sk` columns.

- `update: DynamoDBUpdateObject<T>`

  An object that specifies the attributes to be updated along with the new values for them. The update object can be used with `add`, `remove`, `replace`, `increment`, `decrement`, `append`, `prepend`, `updateListItem`.

Amazon RDS module functions

Amazon RDS module functions provide an enhanced experience when interacting with databases configured with the Amazon RDS Data API. The module is imported using `@aws-appsync/utils/rds`:

```
import * as rds from '@aws-appsync/utils/rds';
```

Functions can also be imported individually. For instance, the import below uses `sql`:

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

Select

The `select` utility creates a SELECT statement to query your relational database.

Basic use

In its basic form, you can specify the table you want to query.

```
import { select, createPgStatement } from '@aws-appsync/utils/rds';

export const onPublish = {
  request(ctx) {
    // Generates statement:
    // "SELECT * FROM "persons"
```

```
    return createPgStatement(select({table: 'persons'}));
  }
}
```

You can also specify the schema in your table identifier:.

```
import { select, createPgStatement } from '@aws-appsync/utils/rds';

export const onPublish = {
  request(ctx) {
    // Generates statement:
    // SELECT * FROM "private"."persons"
    return createPgStatement(select({table: 'private.persons'}));
  }
}
```

Specifying columns

You can specify columns with the `columns` property. If this isn't set to a value, it defaults to `*`.

```
export const onPublish = {
  request(ctx) {
    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name']
    }));
  }
}
```

You can also specify a column's table.

```
export const onPublish = {
  request(ctx) {
    // Generates statement:
    // SELECT "id", "persons"."name"
    // FROM "persons"
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'persons.name']
```

```
    }));
  }
}
```

Limits and offsets

You can apply `limit` and `offset` to the query.

```
export const onPublish = {
  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
    }));
  }
}
```

Order By

You can sort your results with the `orderBy` property. Provide an array of objects specifying the column and an optional `dir` property.

```
export const onPublish = {
  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'}]
    }));
  }
}
```

Filters

You can build filters by using the special condition object.

```
export const onPublish = {
  request(ctx) {
    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    // WHERE "name" = :NAME
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        where: {name: {eq: 'Stephane'}}
    }));
  }
}
```

You can also combine filters.

```
export const onPublish = {
  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}}
    }));
  }
}
```

You can create OR statements.

```
export const onPublish = {
  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 } }
        ]}
    }));
  }
}
```

You can negate a condition with `not`.

```
export const onPublish = {
  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 } }
        ]}
    }));
  }
}
```

You can also use the following operators to compare values:

Operator	Description	Possible value types
eq	Equal	number, string, boolean
ne	Not equal	number, string, boolean
le	Less than or equal	number, string
lt	Less than	number, string
ge	Greater than or equal	number, string
gt	Greater than	number, string

contains	Like	string
notContains	Not like	string
beginsWith	Starts with prefix	string
between	Between two values	number, string
attributeExists	The attribute is not null	number, string, boolean
size	checks the length of the element	string

Insert

The `insert` utility provides a straightforward way of inserting single row items in your database with the `INSERT` operation.

Single item insertions

To insert an item, specify the table and then pass in your object of values. The object keys are mapped to your table columns. Columns names are automatically escaped, and values are sent to the database using the variable map.

```
import { insert, createMySQLStatement } from '@aws-appsync/utils/rds';

export const onPublish = {
  request(ctx) {
    const { input: values } = ctx.args;
    const insertStatement = insert({ table: 'persons', values });

    // Generates statement:
    // INSERT INTO `persons`(`name`)
    // VALUES(:NAME)
    return createMySQLStatement(insertStatement);
  }
}
```

MySQL use case

You can combine an `insert` followed by a `select` to retrieve your inserted row.

```
import { insert, select, createMySQLStatement } from '@aws-appsync/utils/rds';

export const onPublish = {
  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);
  }
}
```

Postgres use case

With Postgres, you can use `returning` to obtain data from the row that you inserted. It accepts * or an array of column names:

```
import { insert, createPgStatement } from '@aws-appsync/utils/rds';

export const onPublish = {
  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);
  }
}
```

Update

The `update` utility allows you to update existing rows. You can use the condition object to apply changes to the specified columns in all the rows that satisfy the condition. For example, let's presume that we have a schema that allows us to make this mutation. The following example updates the `name` of `Person` with the `id` value of `3` but only if we've known them (`known_since`) since the year `2000`.

```
mutation Update {
    updatePerson(
        input: {id: 3, name: "Jon"},
        condition: {known_since: {ge: "2000"}}
    ) {
    id
    name
  }
}
```

Our update handler looks like the following:

```
import { update, createPgStatement } from '@aws-appsync/utils/rds';

export const onPublish = {
  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);
  }
}
```

We can add a check to our condition to make sure that only the row that has the primary key `id` equal to `3` is updated. Similarly, for Postgres `inserts`, you can use `returning` to return the modified data.

Remove

The `remove` utility allows you to delete existing rows. You can use the condition object on all rows that satisfy the condition. Note that `delete` is a reserved keyword in JavaScript. Use `remove` instead.

```
import { remove, createPgStatement } from '@aws-appsync/utils/rds';

export const onPublish = {
  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 some cases, you might require more specificity about the correct object type to use in your statement. You can use the provided type hints to specify the type of your parameters. AWS

AppSync supports the same type hints as the Data API. You can cast your parameters by using the `typeHint` functions from the AWS AppSync `rds` module.

The following example allows you to send an array as a value that is casted as a JSON object. We use the `->` operator to retrieve the element at the `index 2` in the JSON array.

```
import { sql, createPgStatement, toJsonObject, typeHint } from '@aws-appsync/utils/
rds';

export const onPublish = {
  request(ctx) {
    const arr = ctx.args.list_of_ids
    const statement = sql`select ${typeHint.JSON(arr)}->2 as value`
    return createPgStatement(statement)
  }
}
```

Casting is also useful when handling and comparing `DATE`, `TIME`, and `TIMESTAMP`:

```
import { select, createPgStatement, typeHint } from '@aws-appsync/utils/rds';

export const onPublish = {
  request(ctx) {
    const when = ctx.args.when
    const statement = select({
        table: 'persons',
        where: { createdAt : { gt: typeHint.DATETIME(when) } }
    })
    return createPgStatement(statement)
  }
}
```

The following example demonstrates how to send the current date and time.

```
import { sql, createPgStatement, typeHint } from '@aws-appsync/utils/rds';

export const onPublish = {
  request(ctx) {
    const now = util.time.nowFormatted('YYYY-MM-dd HH:mm:ss')
    return createPgStatement(sql`select ${typeHint.TIMESTAMP(now)}`)
  }
}
```

Available type hints

- `typeHint.DATE` — The corresponding parameter is sent as an object of the `DATE` type to the database. The accepted format is `YYYY-MM-DD`.
- `typeHint.DECIMAL` — The corresponding parameter is sent as an object of the `DECIMAL` type to the database.
- `typeHint.JSON` — The corresponding parameter is sent as an object of the `JSON` type to the database.
- `typeHint.TIME` — The corresponding string parameter value is sent as an object of the `TIME` type to the database. The accepted format is `HH:MM:SS[.FFF]`.
- `typeHint.TIMESTAMP` — The corresponding string parameter value is sent as an object of the `TIMESTAMP` type to the database. The accepted format is `YYYY-MM-DD HH:MM:SS[.FFF]`.
- `typeHint.UUID` — The corresponding string parameter value is sent as an object of the `UUID` type to the database.

Runtime utilities

The runtime library provides utilities to control or modify the runtime properties of your handlers and functions.

Invoking the following function stops the execution of the current handler (AWS AppSync Events API) and returns the specified object as the result.

`runtime.earlyReturn(obj?: unknown): never`

When this function is called in an AWS AppSync Events handler, the data source and response function are skipped.

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export const onPublish = {
  request(ctx) {
    if (condition === true) {
      return runtime.earlyReturn(ctx.events)
    }
    // never executed if `condition` is true
    return ddb.batchPut({
      tables: {
        messages: ctx.events.map(({ id, payload }) => ({
```

```
          channel: ctx.info.channelNamespace.name,
          id,
          ...payload
        })),
      }
    });
  },
  // never called if `condition` was true
  response: (ctx) => ctx.events
}
```

AWS AppSync JavaScript function reference for DynamoDB

The Amazon DynamoDB functions allow you to use JavaScript to store and retrieve data in existing Amazon DynamoDB tables in your account. This section describes the request and response handlers for supported DynamoDB operations.

- GetItem — The GetItem request lets you tell the DynamoDB function to make a GetItem request to DynamoDB, and enables you to specify the key of the item in DynamoDB and whether to use a consistent read.
- PutItem — The PutItem request lets you tell the DynamoDB function to make a PutItem request to DynamoDB, and enables you to specify the key of the item in DynamoDB, the full contents of the item (composed of key and attributeValues), and conditions for the operation to succeed.
- UpdateItem — The UpdateItem request enables you to tell the DynamoDB function to make a UpdateItem request to DynamoDB and allows you to specify the key of the item in DynamoDB, an update expression describing how to update the item in DynamoDB, and conditions for the operation to succeed.
- DeleteItem — The DeleteItem request lets you tell the DynamoDB function to make a DeleteItem request to DynamoDB, and enables you to specify the key of the item in DynamoDB and conditions for the operation to succeed.
- Query — The Query request object lets you tell the handler to make a Query request to DynamoDB, and enables you to specify the key expression, which index to use, additional filters, how many items to return, whether to use consistent reads, query direction (forward or backward), and pagination tokens.
- Scan — The Scan request lets you tell the DynamoDB function to make a Scan request to DynamoDB, and enables you to specify a filter to exclude results, which index to use, how many items to return, whether to use consistent reads, pagination tokens, and parallel scans.

- BatchGetItem — The BatchGetItem request object lets you tell the DynamoDB function to make a BatchGetItem request to DynamoDB to retrieve multiple items, potentially across multiple tables. For this request object, you must specify the table names to retrieve the items from and the keys of the items to retrieve from each table.
- BatchDeleteItem — The BatchDeleteItem request object lets you tell the DynamoDB function to make a BatchWriteItem request to DynamoDB to delete multiple items, potentially across multiple tables. For this request object, you must specify the table names to delete the items from and the keys of the items to delete from each table.
- BatchPutItem — The BatchPutItem request object lets you tell the DynamoDB function to make a BatchWriteItem request to DynamoDB to put multiple items, potentially across multiple tables. For this request object, you must specify the table names to put the items in and the full items to put in each table.
- TransactGetItems — The TransactGetItems request object lets you to tell the DynamoDB function to make a TransactGetItems request to DynamoDB to retrieve multiple items, potentially across multiple tables. For this request object, you must specify the table name of each request item to retrieve the item from and the key of each request item to retrieve from each table.
- TransactWriteItems — The TransactWriteItems request object lets you tell the DynamoDB function to make a TransactWriteItems request to DynamoDB to write multiple items, potentially to multiple tables. For this request object, you must specify the destination table name of each request item, the operation of each request item to perform, and the key of each request item to write.
- Type system (request mapping) — Learn more about how DynamoDB typing is integrated into AWS AppSync requests.
- Type system (response mapping) — Learn more about how DynamoDB types are converted automatically to JSON in a response payload.
- Filters — Learn more about filters for query and scan operations.
- Condition expressions — Learn more about condition expressions for PutItem, UpdateItem, and DeleteItem operations.
- Transaction condition expressions — Learn more about condition expressions for TransactWriteItems operations.
- Projections — Learn more about how to specify attributes in read operations.

GetItem

> **Note**
>
> We recommend using the DynamoDB built-in module to generate your request. For more information, see Amazon DynamoDB built-in module.

The `GetItem` request lets you tell the AWS AppSync DynamoDB function to make a `GetItem` request to DynamoDB, and enables you to specify:

- The key of the item in DynamoDB
- Whether to use a consistent read or not

The `GetItem` request has the following structure:

```
type DynamoDBGetItem = {
  operation: 'GetItem';
  key: { [key: string]: any };
  consistentRead?: ConsistentRead;
  projection?: {
    expression: string;
    expressionNames?: { [key: string]: string };
  };
};
```

The TypeScript definition above shows all available fields for the request. While you can construct this request manually, using the DynamoDB built-in module is the recommended approach for generating accurate and efficient requests.

GetItem fields

`operation`

The DynamoDB operation to perform. To perform the `GetItem` DynamoDB operation, this must be set to `GetItem`. This value is required.

`key`

The key of the item in DynamoDB. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about how to specify a "typed value", see Type system (request mapping). This value is required.

`consistentRead`

Whether or not to perform a strongly consistent read with DynamoDB. This is optional, and defaults to `false`.

`projection`

A projection that's used to specify the attributes to return from the DynamoDB operation. For more information about projections, see Projections. This field is optional.

For more information about DynamoDB type conversion, see Type system (response mapping).

Examples

```
export const onPublish = {
  request: (ctx) => ({
    operation : "GetItem",
    key : util.dynamodb.toMapValues({
      channel: ctx.info.channelNamespace.name,
      id: ctx.events[0].payload.id}),
    consistentRead : true
  }),
  response(ctx) {
    return [{
      id: ctx.event[0].id,
      payload: ctx.result
    }]
  }
}
```

The following example demonstrates DynamoDB utils.

```
import * as ddb from '@aws-appsync/utils/dynamodb'
export const onPublish = {
  request: (ctx) => ddb.get({
    key: {
```

```
      channel: ctx.info.channelNamespace.name,
      id: ctx.events[0].payload.id
    },
    consistentRead: true
  }),
  response(ctx) {
    return [{
      id: ctx.event[0].id,
      payload: ctx.result
    }]
  }
}
```

For more information about the DynamoDB `GetItem` API, see the DynamoDB API documentation.

PutItem

> **Note**
>
> We recommend using the DynamoDB built-in module to generate your request. For more information, see Amazon DynamoDB built-in module.

The `PutItem` request enables you to create or replace items in DynamoDB through AWS AppSync. The request specifies the following:

- Item Key: The unique identifier for the DynamoDB item
- Item Contents: The complete item data, including both the `key` and `attributeValues`
- Operation Conditions (optional): Rules that must be met for the operation to proceed

The `PutItem` request has the following structure:

```
type DynamoDBPutItemRequest = {
  operation: 'PutItem';
  key: { [key: string]: any };
  attributeValues: { [key: string]: any};
  condition?: ConditionCheckExpression;
  customPartitionKey?: string;
  populateIndexFields?: boolean;
  _version?: number;
```

```
};
```

The TypeScript definition above shows all available fields for the request. While you can construct this request manually, we recommend using the DynamoDB built-in module for generating accurate and efficient requests.

PutItem fields

`operation`

The DynamoDB operation to perform. To perform the `PutItem` DynamoDB operation, this must be set to `PutItem`. This value is required.

`key`

The key of the item in DynamoDB. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about how to specify a "typed value", see Type system (request mapping). This value is required.

`attributeValues`

The rest of the attributes of the item to be put into DynamoDB. For more information about how to specify a "typed value", see Type system (request mapping). This field is optional.

`condition`

A condition to determine if the request should succeed or not, based on the state of the object already in DynamoDB. If no condition is specified, the `PutItem` request overwrites any existing entry for that item. For more information about conditions, see Condition expressions. This value is optional.

`_version`

A numeric value that represents the latest known version of an item. This value is optional. This field is used for *Conflict Detection* and is only supported on versioned data sources.

`customPartitionKey`

When enabled, this string value modifies the format of the `ds_sk` and `ds_pk` records used by the delta sync table when versioning has been enabled (for more information, see Conflict detection and sync in the *AWS AppSync Developer Guide*). When enabled, the processing of the `populateIndexFields` entry is also enabled. This field is optional.

Not supported in AWS AppSync Events

`populateIndexFields`

A boolean value that, when enabled **along with the `customPartitionKey`**, creates new entries for each record in the delta sync table, specifically in the `gsi_ds_pk` and `gsi_ds_sk` columns. For more information, see Conflict detection and sync in the *AWS AppSync Developer Guide*. This field is optional.

The item written to DynamoDB is automatically converted to JSON primitive types and is available in the context result (`context.result`).

For more information about DynamoDB type conversion, see Type system (response mapping).

For more information about the DynamoDB `PutItem` API, see the DynamoDB API documentation.

UpdateItem

> **Note**
>
> We recommend using the DynamoDB built-in module to generate your request. For more information, see Amazon DynamoDB built-in module.

The `UpdateItem` request enables you to modify existing items in DynamoDB through AWS AppSync. The request specifies the following:

- Item Key: The unique identifier for the DynamoDB item to update
- Item Expression: Describes how to modify the item in DynamoDB
- Operation Conditions (optional): Rules that must be met for the update to proceed

The `UpdateItem` request has the following structure:

```
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;
};
```

The TypeScript definition above shows all available fields for the request. While you can construct this request manually, we recommend using the DynamoDB built-in module for generating accurate and efficient requests.

UpdateItem fields

`operation`

The DynamoDB operation to perform. To perform the `UpdateItem` DynamoDB operation, this must be set to `UpdateItem`. This value is required.

`key`

The key of the item in DynamoDB. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about specifying a "typed value", see Type system (request mapping). This value is required.

`update`

The `update` section lets you specify an update expression that describes how to update the item in DynamoDB. For more information about how to write update expressions, see the DynamoDB UpdateExpressions documentation. This section is required.

The `update` section has three components:

`expression`

The update expression. This value is required.

`expressionNames`

The substitutions for expression attribute *name* placeholders, in the form of key-value pairs. The key corresponds to a name placeholder used in the `expression`, and the value must be a string corresponding to the attribute name of the item in DynamoDB. This field is optional,

and should only be populated with substitutions for expression attribute name placeholders used in the `expression`.

`expressionValues`

The substitutions for expression attribute *value* placeholders, in the form of key-value pairs. The key corresponds to a value placeholder used in the `expression`, and the value must be a typed value. For more information about how to specify a "typed value", see Type system (request mapping). This must be specified. This field is optional, and should only be populated with substitutions for expression attribute value placeholders used in the `expression`.

`condition`

A condition to determine if the request should succeed or not, based on the state of the object already in DynamoDB. If no condition is specified, the `UpdateItem` request updates the existing entry regardless of its current state. For more information about conditions, see Condition expressions. This value is optional.

`_version`

A numeric value that represents the latest known version of an item. This value is optional. This field is used for *Conflict Detection* and is only supported on versioned data sources.

Not supported in AWS AppSync Events

`customPartitionKey`

When enabled, this string value modifies the format of the `ds_sk` and `ds_pk` records used by the delta sync table when versioning has been enabled (for more information, see Conflict detection and sync in the *AWS AppSync Developer Guide*). When enabled, the processing of the `populateIndexFields` entry is also enabled. This field is optional.

Not supported in AWS AppSync Events

`populateIndexFields`

A boolean value that, when enabled **along with the `customPartitionKey`**, creates new entries for each record in the delta sync table, specifically in the `gsi_ds_pk` and `gsi_ds_sk` columns. For more information, see Conflict detection and sync in the *AWS AppSync Developer Guide*. This field is optional.

Not supported in AWS AppSync Events

DeleteItem

> **Note**
>
> We recommend using the DynamoDB built-in module to generate your request. For more information, see Amazon DynamoDB built-in module.

The `DeleteItem` request enables you to delete an item in a DynamoDB table. The request specifies the following:

- The key of the item in DynamoDB
- Conditions for the operation to succeed

The `DeleteItem` request has the following structure:

```
type DynamoDBDeleteItemRequest = {
  operation: 'DeleteItem';
  key: { [key: string]: any };
  condition?: ConditionCheckExpression;
  customPartitionKey?: string;
  populateIndexFields?: boolean;
  _version?: number;
};
```

The TypeScript definition above shows all available fields for the request. While you can construct this request manually, we recommend using the DynamoDB built-in module for generating accurate and efficient requests.

DeleteItem fields

`operation`

The DynamoDB operation to perform. To perform the `DeleteItem` DynamoDB operation, this must be set to `DeleteItem`. This value is required.

`key`

The key of the item in DynamoDB. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about specifying a "typed value", see Type system (request mapping). This value is required.

`condition`

A condition to determine if the request should succeed or not, based on the state of the object already in DynamoDB. If no condition is specified, the `DeleteItem` request deletes an item regardless of its current state. For more information about conditions, see Condition expressions. This value is optional.

`_version`

A numeric value that represents the latest known version of an item. This value is optional. This field is used for *Conflict Detection* and is only supported on versioned data sources.

Not supported in AWS AppSync Events

`customPartitionKey`

When enabled, this string value modifies the format of the `ds_sk` and `ds_pk` records used by the delta sync table when versioning has been enabled (for more information, see Conflict detection and sync in the *AWS AppSync Developer Guide*). When enabled, the processing of the `populateIndexFields` entry is also enabled. This field is optional.

Not supported in AWS AppSync Events

`populateIndexFields`

A boolean value that, when enabled **along with the `customPartitionKey`**, creates new entries for each record in the delta sync table, specifically in the `gsi_ds_pk` and `gsi_ds_sk` columns. For more information, see Conflict detection and sync in the *AWS AppSync Developer Guide*. This field is optional.

Not supported in AWS AppSync Events

For more information about the DynamoDB `DeleteItem` API, see the DynamoDB API documentation.

Query

> **Note**
>
> We recommend using the DynamoDB built-in module to generate your request. For more information, see Amazon DynamoDB built-in module.

The `Query` request enables you to efficiently select all items in a DynamoDB table that match a key condition. The request specifies the following:

- Key expression
- Which index to use
- Any additional filter
- How many items to return
- Whether to use consistent reads
- Query direction (forward or backward)
- Pagination token

The `Query` request object has the following structure:

```
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 };
  };
};
```

The TypeScript definition above shows all available fields for the request. While you can construct this request manually, we recommend using the DynamoDB built-in module for generating accurate and efficient requests.

Query fields

`operation`

The DynamoDB operation to perform. To perform the `Query` DynamoDB operation, this must be set to `Query`. This value is required.

`query`

The `query` section lets you specify a key condition expression that describes which items to retrieve from DynamoDB. For more information about how to write key condition expressions, see the DynamoDB KeyConditions documentation . This section must be specified.

`expression`

The query expression. This field must be specified.

`expressionNames`

The substitutions for expression attribute *name* placeholders, in the form of key-value pairs. The key corresponds to a name placeholder used in the `expression`, and the value must be a string corresponding to the attribute name of the item in DynamoDB. This field is optional, and should only be populated with substitutions for expression attribute name placeholders used in the `expression`.

`expressionValues`

The substitutions for expression attribute *value* placeholders, in the form of key-value pairs. The key corresponds to a value placeholder used in the `expression`, and the value must be a typed value. For more information about how to specify a “typed value”, see Type system (request mapping). This value is required. This field is optional, and should only be populated with substitutions for expression attribute value placeholders used in the `expression`.

`filter`

An additional filter that can be used to filter the results from DynamoDB before they are returned. For more information about filters, see Filters. This field is optional.

`index`

The name of the index to query. The DynamoDB query operation allows you to scan on Local Secondary Indexes and Global Secondary Indexes in addition to the primary key index for a hash key. If specified, this tells DynamoDB to query the specified index. If omitted, the primary key index is queried.

`nextToken`

The pagination token to continue a previous query. This would have been obtained from a previous query. This field is optional.

`limit`

The maximum number of items to evaluate (not necessarily the number of matching items). This field is optional.

`scanIndexForward`

A boolean indicating whether to query forwards or backwards. This field is optional, and defaults to `true`.

`consistentRead`

A boolean indicating whether to use consistent reads when querying DynamoDB. This field is optional, and defaults to `false`.

`select`

By default, the AWS AppSync DynamoDB resolver only returns attributes that are projected into the index. If more attributes are required, you can set this field. This field is optional. The supported values are:

`ALL_ATTRIBUTES`

Returns all of the item attributes from the specified table or index. If you query a local secondary index, DynamoDB fetches the entire item from the parent table for each matching item in the index. If the index is configured to project all item attributes, all of the data can be obtained from the local secondary index and no fetching is required.

`ALL_PROJECTED_ATTRIBUTES`

Allowed only when querying an index. Retrieves all attributes that have been projected into the index. If the index is configured to project all attributes, this return value is equivalent to specifying `ALL_ATTRIBUTES`.

`SPECIFIC_ATTRIBUTES`

Returns only the attributes listed in the `projection`'s `expression`. This return value is equivalent to specifying the `projection`'s `expression` without specifying any value for `Select`.

`projection`

A projection that's used to specify the attributes to return from the DynamoDB operation. For more information about projections, see Projections. This field is optional.

For more information about the DynamoDB `Query` API, see the DynamoDB API documentation.

Scan

> **Note**
>
> We recommend using the DynamoDB built-in module to generate your request. For more information, see Amazon DynamoDB built-in module.

The `Scan` request scans for items across a DynamoDB table. The request specifies the following:

- A filter to exclude results
- Which index to use
- How many items to return
- Whether to use consistent reads
- Pagination token
- Parallel scans

The `Scan` request object has the following structure:

```
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 };
  };
};
```

The TypeScript definition above shows all available fields for the request. While you can construct this request manually, we recommend using the DynamoDB built-in module for generating accurate and efficient requests.

Scan fields

`operation`

The DynamoDB operation to perform. To perform the `Scan` DynamoDB operation, this must be set to `Scan`. This value is required.

`filter`

A filter that can be used to filter the results from DynamoDB before they are returned. For more information about filters, see Filters. This field is optional.

`index`

The name of the index to query. The DynamoDB query operation allows you to scan on Local Secondary Indexes and Global Secondary Indexes in addition to the primary key index for a hash key. If specified, this tells DynamoDB to query the specified index. If omitted, the primary key index is queried.

`limit`

The maximum number of items to evaluate at a single time. This field is optional.

consistentRead

A Boolean that indicates whether to use consistent reads when querying DynamoDB. This field is optional, and defaults to `false`.

nextToken

The pagination token to continue a previous query. This would have been obtained from a previous query. This field is optional.

select

By default, the AWS AppSync DynamoDB function only returns whatever attributes are projected into the index. If more attributes are required, then this field can be set. This field is optional. The supported values are:

ALL_ATTRIBUTES

Returns all of the item attributes from the specified table or index. If you query a local secondary index, DynamoDB fetches the entire item from the parent table for each matching item in the index. If the index is configured to project all item attributes, all of the data can be obtained from the local secondary index and no fetching is required.

ALL_PROJECTED_ATTRIBUTES

Allowed only when querying an index. Retrieves all attributes that have been projected into the index. If the index is configured to project all attributes, this return value is equivalent to specifying `ALL_ATTRIBUTES`.

SPECIFIC_ATTRIBUTES

Returns only the attributes listed in the `projection`'s `expression`. This return value is equivalent to specifying the `projection`'s `expression` without specifying any value for `Select`.

totalSegments

The number of segments to partition the table by when performing a parallel scan. This field is optional, but must be specified if `segment` is specified.

segment

The table segment in this operation when performing a parallel scan. This field is optional, but must be specified if `totalSegments` is specified.

`projection`

A projection that's used to specify the attributes to return from the DynamoDB operation. For more information about projections, see Projections. This field is optional.

The results have the following structure:

```
{
    items = [ ... ],
    nextToken = "a pagination token",
    scannedCount = 10
}
```

The fields are defined as follows:

`items`

A list containing the items returned by the DynamoDB scan.

`nextToken`

If there might be more results, `nextToken` contains a pagination token that you can use in another request. AWS AppSync encrypts and obfuscates the pagination token returned from DynamoDB. This prevents your table data from being inadvertently leaked to the caller. Also, these pagination tokens can't be used across different functions.

`scannedCount`

The number of items that were retrieved by DynamoDB before a filter expression (if present) was applied.

For more information about the DynamoDB `Scan` API, see the DynamoDB API documentation.

BatchGetItem

> **Note**
>
> We recommend using the DynamoDB built-in module to generate your request. For more information, see Amazon DynamoDB built-in module.

The `BatchGetItem` request object enables you to retrieve multiple items, potentially across multiple DynamoDB tables. For this request object, you must specify the following:

- The names of the table to retrieve the items from
- The keys of the items to retrieve from each table

The DynamoDB `BatchGetItem` limits apply and **no condition expression** can be provided.

The `BatchGetItem` request object has the following structure:

```
type DynamoDBBatchGetItemRequest = {
  operation: 'BatchGetItem';
  tables: {
    [tableName: string]: {
      keys: { [key: string]: any }[];
      consistentRead?: boolean;
      projection?: {
        expression: string;
        expressionNames?: { [key: string]: string };
      };
    };
  };
};
```

The TypeScript definition above shows all available fields for the request. While you can construct this request manually, we recommend using the DynamoDB built-in module for generating accurate and efficient requests.

BatchGetItem fields

`operation`

The DynamoDB operation to perform. To perform the `BatchGetItem` DynamoDB operation, this must be set to `BatchGetItem`. This value is required.

`tables`

The DynamoDB tables to retrieve the items from. The value is a map where table names are specified as the keys of the map. At least one table must be provided. This `tables` value is required.

`keys`

List of DynamoDB keys representing the primary key of the items to retrieve. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about how to specify a “typed value”, see Type system (request mapping).

`consistentRead`

Whether to use a consistent read when executing a *GetItem* operation. This value is optional and defaults to *false*.

`projection`

A projection that's used to specify the attributes to return from the DynamoDB operation. For more information about projections, see Projections. This field is optional.

Things to remember:

- If an item has not been retrieved from the table, a *null* element appears in the data block for that table.
- Invocation results are sorted per table, based on the order in which they were provided inside the request object.
- Each `Get` command inside a `BatchGetItem` is atomic, however, a batch can be partially processed. If a batch is partially processed due to an error, the unprocessed keys are returned as part of the invocation result inside the *unprocessedKeys* block.
- `BatchGetItem` is limited to 100 keys.

Response structure

```
type Response = {
  data: {
    [tableName: string]: {[key: string]: any}[]
  }
  unprocessedKeys: {
    [tableName: string]: {[key: string]: string}[]
  }
}
```

BatchDeleteItem

> **Note**
>
> We recommend using the DynamoDB built-in module to generate your request. For more information, see Amazon DynamoDB built-in module.

The `BatchDeleteItem` request deletes multiple items, potentially across multiple tables using a `BatchWriteItem` request. The request specifies the following:

- The names of the tables to delete the items from
- The keys of the items to delete from each table

The DynamoDB `BatchWriteItem` limits apply and **no condition expression** can be provided.

The `BatchDeleteItem` request object has the following structure:

```
type DynamoDBBatchDeleteItemRequest = {
  operation: 'BatchDeleteItem';
  tables: {
    [tableName: string]: { [key: string]: any }[];
  };
};
```

The TypeScript definition above shows all available fields for the request. While you can construct this request manually, we recommend using the DynamoDB built-in module for generating accurate and efficient requests.

BatchDeleteItem fields

`operation`

The DynamoDB operation to perform. To perform the `BatchDeleteItem` DynamoDB operation, this must be set to `BatchDeleteItem`. This value is required.

`tables`

The DynamoDB tables to delete the items from. Each table is a list of DynamoDB keys representing the primary key of the items to delete. DynamoDB items may have a single hash

key, or a hash key and sort key, depending on the table structure. For more information about how to specify a "typed value", see Type system (request mapping). At least one table must be provided. The `tables` value is required.

Things to remember:

- Contrary to the `DeleteItem` operation, the fully deleted item isn't returned in the response. Only the passed key is returned.
- If an item has not been deleted from the table, a *null* element appears in the data block for that table.
- Invocation results are sorted per table, based on the order in which they were provided inside the request object.
- Each `Delete` command inside a `BatchDeleteItem` is atomic. However a batch can be partially processed. If a batch is partially processed due to an error, the unprocessed keys are returned as part of the invocation result inside the *unprocessedKeys* block.
- `BatchDeleteItem` is limited to 25 keys.
- This operation **is not** supported when used with conflict detection. Using both at the same time may result in an error.

Response structure (in `ctx.result`)

```
type Response = {
  data: {
    [tableName: string]: {[key: string]: any}[]
  }
  unprocessedKeys: {
    [tableName: string]: {[key: string]: any}[]
  }
}
```

The `ctx.error` contains details about the error. The keys **data**, **unprocessedKeys**, and each table key that was provided in the function request object are guaranteed to be present in the invocation result. Items that have been deleted are present in the **data** block. Items that haven't been processed are marked as *null* inside the data block and are placed inside the **unprocessedKeys** block.

BatchPutItem

> **Note**
>
> We recommend using the DynamoDB built-in module to generate your request. For more information, see Amazon DynamoDB built-in module.

The `BatchPutItem` request enables you to put multiple items, potentially across multiple DynamoDB tables using a `BatchWriteItem` request. The request specifies the following:

- The names of the tables to put the items in
- The full list of items to put in each table

The DynamoDB `BatchWriteItem` limits apply and **no condition expression** can be provided.

The `BatchPutItem` request object has the following structure:

```
type DynamoDBBatchPutItemRequest = {
  operation: 'BatchPutItem';
  tables: {
    [tableName: string]: { [key: string]: any}[];
  };
};
```

The TypeScript definition above shows all available fields for the request. While you can construct this request manually, we recommend using the DynamoDB built-in module for generating accurate and efficient requests.

BatchPutItem fields

`operation`

The DynamoDB operation to perform. To perform the `BatchPutItem` DynamoDB operation, this must be set to `BatchPutItem`. This value is required.

`tables`

The DynamoDB tables to put the items in. Each table entry represents a list of DynamoDB items to insert for this specific table. At least one table must be provided. This value is required.

Things to remember:

- The fully inserted items are returned in the response, if successful.
- If an item hasn't been inserted in the table, a *null* element is displayed in the data block for that table.
- The inserted items are sorted per table, based on the order in which they were provided inside the request object.
- Each `Put` command inside a `BatchPutItem` is atomic, however, a batch can be partially processed. If a batch is partially processed due to an error, the unprocessed keys are returned as part of the invocation result inside the *unprocessedKeys* block.
- `BatchPutItem` is limited to 25 items.
- This operation **is not** supported when used with conflict detection. Using both at the same time may result in an error.

Response structure (in `ctx.result`)

```
type Response = {
  data: {
    [tableName: string]: {[key: string]: any}[]
  }
  unprocessedItems: {
    [tableName: string]: {[key: string]: any}[]
  }
}
```

The `ctx.error` contains details about the error. The keys **data**, **unprocessedItems**, and each table key that was provided in the request object are guaranteed to be present in the invocation result. Items that have been inserted are in the **data** block. Items that haven't been processed are marked as *null* inside the data block and are placed inside the **unprocessedItems** block.

TransactGetItems

> **Note**
>
> We recommend using the DynamoDB built-in module to generate your request. For more information, see Amazon DynamoDB built-in module.

The `TransactGetItems` request object retrieves multiple items, potentially across multiple DynamoDB tables in a single transaction. The request specifies the following:

- The names of the tables to retrieve each item from
- The key of each request item to retrieve from each table

The DynamoDB `TransactGetItems` limits apply and **no condition expression** can be provided.

The `TransactGetItems` request object has the following structure:

```
type DynamoDBTransactGetItemsRequest = {
  operation: 'TransactGetItems';
  transactItems: { table: string; key: { [key: string]: any }; projection?:
 { expression: string; expressionNames?: { [key: string]: string }; }[];
  };
};
```

The TypeScript definition above shows all available fields for the request. While you can construct this request manually, we recommend using the DynamoDB built-in module for generating accurate and efficient requests.

TransactGetItems fields

`operation`

The DynamoDB operation to perform. To perform the `TransactGetItems` DynamoDB operation, this must be set to `TransactGetItems`. This value is required.

`transactItems`

The request items to include. The value is an array of request items. At least one request item must be provided. This `transactItems` value is required.

`table`

The DynamoDB table to retrieve the item from. The value is a string of the table name. This `table` value is required.

`key`

The DynamoDB key representing the primary key of the item to retrieve. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about how to specify a “typed value”, see Type system (request mapping).

`projection`

A projection that's used to specify the attributes to return from the DynamoDB operation. For more information about projections, see Projections. This field is optional.

Things to remember:

- If a transaction succeeds, the order of retrieved items in the `items` block will be the same as the order of request items.
- Transactions are performed in an all-or-nothing way. If any request item causes an error, the whole transaction will not be performed and error details will be returned.
- A request item being unable to be retrieved is not an error. Instead, a *null* element appears in the *items* block in the corresponding position.
- If the error of a transaction is *TransactionCanceledException*, the `cancellationReasons` block will be populated. The order of cancellation reasons in `cancellationReasons` block will be the same as the order of request items.
- `TransactGetItems` is limited to 100 request items.

Response structure (in `ctx.result`)

```
type Response = {
  items?: ({[key: string]: any} | null)[];
  cancellationReasons?: {
    type: string;
    message: string;
  }[]
}
```

The `ctx.error` contains details about the error. The keys **items** and **cancellationReasons** are guaranteed to be present in `ctx.result`.

TransactWriteItems

> **Note**
>
> We recommend using the DynamoDB built-in module to generate your request. For more information, see Amazon DynamoDB built-in module.

The `TransactWriteItems` request writes multiple items, potentially to multiple DynamoDB tables. The request specifies the following:

- The destination table name of each request item
- The operation to perform for each request item. There are four types of operations that are supported: *PutItem*, *UpdateItem*, *DeleteItem*, and *ConditionCheck*
- The key of each request item to write

The DynamoDB `TransactWriteItems` limits apply.

The `TransactWriteItems` request object has the following structure:

```
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;
};
```

The TypeScript definition above shows all available fields for the request. While you can construct this request manually, we recommend using the DynamoDB built-in module for generating accurate and efficient requests.

TransactWriteItems fields

The fields are defined as follows:

`operation`

The DynamoDB operation to perform. To perform the `TransactWriteItems` DynamoDB operation, this must be set to `TransactWriteItems`. This value is required.

`transactItems`

The request items to include. The value is an array of request items. At least one request item must be provided. This `transactItems` value is required.

For `PutItem`, the fields are defined as follows:

`table`

The destination DynamoDB table. The value is a string of the table name. This `table` value is required.

`operation`

The DynamoDB operation to perform. To perform the `PutItem` DynamoDB operation, this must be set to `PutItem`. This value is required.

`key`

The DynamoDB key representing the primary key of the item to put. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure.

For more information about how to specify a “typed value”, see Type system (request mapping). This value is required.

`attributeValues`

The rest of the attributes of the item to be put into DynamoDB. For more information about how to specify a “typed value”, see Type system (request mapping). This field is optional.

`condition`

A condition to determine if the request should succeed or not, based on the state of the object already in DynamoDB. If no condition is specified, the `PutItem` request overwrites any existing entry for that item. You can specify whether to retrieve the existing item back when condition check fails. For more information about transactional conditions, see Transaction condition expressions. This value is optional.

For `UpdateItem`, the fields are defined as follows:

`table`

The DynamoDB table to update. The value is a string of the table name. This `table` value is required.

`operation`

The DynamoDB operation to perform. To perform the `UpdateItem` DynamoDB operation, this must be set to `UpdateItem`. This value is required.

`key`

The DynamoDB key representing the primary key of the item to update. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about how to specify a “typed value”, see Type system (request mapping). This value is required.

`update`

The `update` section lets you specify an update expression that describes how to update the item in DynamoDB. For more information about how to write update expressions, see the DynamoDB UpdateExpressions documentation. This section is required.

`condition`

A condition to determine if the request should succeed or not, based on the state of the object already in DynamoDB. If no condition is specified, the `UpdateItem` request

updates the existing entry regardless of its current state. You can specify whether to retrieve the existing item back when condition check fails. For more information about transactional conditions, see Transaction condition expressions. This value is optional.

For `DeleteItem`, the fields are defined as follows:

`table`

The DynamoDB table in which to delete the item. The value is a string of the table name. This `table` value is required.

`operation`

The DynamoDB operation to perform. To perform the `DeleteItem` DynamoDB operation, this must be set to `DeleteItem`. This value is required.

`key`

The DynamoDB key representing the primary key of the item to delete. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about how to specify a “typed value”, see Type system (request mapping). This value is required.

`condition`

A condition to determine if the request should succeed or not, based on the state of the object already in DynamoDB. If no condition is specified, the `DeleteItem` request deletes an item regardless of its current state. You can specify whether to retrieve the existing item back when condition check fails. For more information about transactional conditions, see Transaction condition expressions. This value is optional.

For `ConditionCheck`, the fields are defined as follows:

`table`

The DynamoDB table in which to check the condition. The value is a string of the table name. This `table` value is required.

`operation`

The DynamoDB operation to perform. To perform the `ConditionCheck` DynamoDB operation, this must be set to `ConditionCheck`. This value is required.

key

The DynamoDB key representing the primary key of the item to condition check. DynamoDB items may have a single hash key, or a hash key and sort key, depending on the table structure. For more information about how to specify a “typed value”, see Type system (request mapping). This value is required.

`condition`

A condition to determine if the request should succeed or not, based on the state of the object already in DynamoDB. You can specify whether to retrieve the existing item back when condition check fails. For more information about transactional conditions, see Transaction condition expressions. This value is required.

Things to remember:

- Only keys of request items are returned in the response, if successful. The order of keys will be the same as the order of request items.
- Transactions are performed in an all-or-nothing way. If any request item causes an error, the whole transaction will not be performed and error details will be returned.
- No two request items can target the same item. Otherwise they will cause *TransactionCanceledException* error.
- If the error of a transaction is *TransactionCanceledException*, the `cancellationReasons` block will be populated. If a request item’s condition check fails **and** you did not specify `returnValuesOnConditionCheckFailure` to be `false`, the item existing in the table will be retrieved and stored in `item` at the corresponding position of `cancellationReasons` block.
- `TransactWriteItems` is limited to 100 request items.
- This operation **is not** supported when used with conflict detection. Using both at the same time may result in an error.

Response structure (in `ctx.result`)

```
type Responser = {
  keys?: {[key: string]: string}[];
  cancellationReasons?: {
    item?: { [key: string]: any };
    type: string;
    message;
```

```
  }
}
```

The `ctx.error` contains details about the error. The keys **keys** and **cancellationReasons** are guaranteed to be present in `ctx.result`.

Type system (request mapping)

When using the AWS AppSync DynamoDB function to call your DynamoDB tables, you must specify your data using the DynamoDB type notation. For more information about DynamoDB data types, see the DynamoDB Data type descriptors and Data types documentation.

> **Note**
>
> You don't have to use DynamoDB type notation when using the DynamoDB built-in module. For more information, see Amazon DynamoDB built-in module.

A DynamoDB value is represented by a JSON object containing a single key-value pair. The key specifies the DynamoDB type, and the value specifies the value itself. In the following example, the key S denotes that the value is a string, and the value `identifier` is the string value itself.

```
{ "S" : "identifier" }
```

The JSON object can't have more than one key-value pair. If more than one key-value pair is specified, the request object isn't parsed.

A DynamoDB value is used anywhere in a request object where you need to specify a value. Some places where you need to do this include: `key` and `attributeValue` sections, and the `expressionValues` section of expression sections. In the following example, the DynamoDB String value `identifier` is being assigned to the `id` field in a `key` section (perhaps in a `GetItem` request object).

```
"key" : {
   "id" : { "S" : "identifier" }
}
```

Supported Types

AWS AppSync supports the following DynamoDB scalar, document, and set types:

String type S

A single string value. A DynamoDB String value is denoted by:

```
{ "S" : "some string" }
```

An example usage is:

```
"key" : {
   "id" : { "S" : "some string" }
}
```

String set type SS

A set of string values. A DynamoDB String Set value is denoted by:

```
{ "SS" : [ "first value", "second value", ... ] }
```

An example usage is:

```
"attributeValues" : {
   "phoneNumbers" : { "SS" : [ "+1 555 123 4567", "+1 555 234 5678" ] }
}
```

Number type N

A single numeric value. A DynamoDB Number value is denoted by:

```
{ "N" : 1234 }
```

An example usage is:

```
"expressionValues" : {
   ":expectedVersion" : { "N" : 1 }
}
```

Number set type NS

A set of number values. A DynamoDB Number Set value is denoted by:

```
{ "NS" : [ 1, 2.3, 4 ... ] }
```

An example usage is:

```
"attributeValues" : {
   "sensorReadings" : { "NS" : [ 67.8, 12.2, 70 ] }
}
```

Binary type B

A binary value. A DynamoDB Binary value is denoted by:

```
{ "B" : "SGVsbG8sIFdvcmxkIQo=" }
```

Note that the value is actually a string, where the string is the base64-encoded representation of the binary data. AWS AppSync decodes this string back into its binary value before sending it to DynamoDB. AWS AppSync uses the base64 decoding scheme as defined by RFC 2045: any character that isn't in the base64 alphabet is ignored.

An example usage is:

```
"attributeValues" : {
   "binaryMessage" : { "B" : "SGVsbG8sIFdvcmxkIQo=" }
}
```

Binary set type BS

A set of binary values. A DynamoDB Binary Set value is denoted by:

```
{ "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ] }
```

Note that the value is actually a string, where the string is the base64-encoded representation of the binary data. AWS AppSync decodes this string back into its binary value before sending it to DynamoDB. AWS AppSync uses the base64 decoding scheme as defined by RFC 2045: any character that is not in the base64 alphabet is ignored.

An example usage is:

```
"attributeValues" : {
```

```
   "binaryMessages" : { "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ] }
}
```

Boolean type BOOL

A Boolean value. A DynamoDB Boolean value is denoted by:

```
{ "BOOL" : true }
```

Note that only `true` and `false` are valid values.

An example usage is:

```
"attributeValues" : {
   "orderComplete" : { "BOOL" : false }
}
```

List type L

A list of any other supported DynamoDB value. A DynamoDB List value is denoted by:

```
{ "L" : [ ... ] }
```

Note that the value is a compound value, where the list can contain zero or more of any supported DynamoDB value (including other lists). The list can also contain a mix of different types.

An example usage is:

```
{ "L" : [
      { "S"  : "A string value" },
      { "N"  : 1 },
      { "SS" : [ "Another string value", "Even more string values!" ] }
   ]
}
```

Map type M

Representing an unordered collection of key-value pairs of other supported DynamoDB values. A DynamoDB Map value is denoted by:

```
{ "M" : { ... } }
```

Note that a map can contain zero or more key-value pairs. The key must be a string, and the value can be any supported DynamoDB value (including other maps). The map can also contain a mix of different types.

An example usage is:

```
{ "M" : {
      "someString" : { "S"  : "A string value" },
      "someNumber" : { "N"  : 1 },
      "stringSet"  : { "SS" : [ "Another string value", "Even more string
 values!" ] }
   }
}
```

Null type `NULL`

A null value. A DynamoDB Null value is denoted by:

```
{ "NULL" : null }
```

An example usage is:

```
"attributeValues" : {
   "phoneNumbers" : { "NULL" : null }
}
```

For more information about each type, see the DynamoDB documentation .

Type system (response mapping)

When receiving a response from DynamoDB, AWS AppSync automatically converts it into JSON primitive types. Each attribute in DynamoDB is decoded and returned in the response handler's context.

For example, if DynamoDB returns the following:

```
{
```

```
    "id" : { "S" : "1234" },
    "name" : { "S" : "Nadia" },
    "age" : { "N" : 25 }
}
```

When the result is returned from your handler, AWS AppSync converts it into a JSON types as:

```
{
    "id" : "1234",
    "name" : "Nadia",
    "age" : 25
}
```

This section explains how AWS AppSync converts the following DynamoDB scalar, document, and set types:

String type S

A single string value. A DynamoDB String value is returned as a string.

For example, if DynamoDB returned the following DynamoDB String value:

```
{ "S" : "some string" }
```

AWS AppSync converts it to a string:

```
"some string"
```

String set type SS

A set of string values. A DynamoDB String Set value is returned as a list of strings.

For example, if DynamoDB returned the following DynamoDB String Set value:

```
{ "SS" : [ "first value", "second value", ... ] }
```

AWS AppSync converts it to a list of strings:

```
[ "+1 555 123 4567", "+1 555 234 5678" ]
```

Number type N

A single numeric value. A DynamoDB Number value is returned as a number.

For example, if DynamoDB returned the following DynamoDB Number value:

```
{ "N" : 1234 }
```

AWS AppSync converts it to a number:

```
1234
```

Number set type NS

A set of number values. A DynamoDB Number Set value is returned as a list of numbers.

For example, if DynamoDB returned the following DynamoDB Number Set value:

```
{ "NS" : [ 67.8, 12.2, 70 ] }
```

AWS AppSync converts it to a list of numbers:

```
[ 67.8, 12.2, 70 ]
```

Binary type B

A binary value. A DynamoDB Binary value is returned as a string containing the base64 representation of that value.

For example, if DynamoDB returned the following DynamoDB Binary value:

```
{ "B" : "SGVsbG8sIFdvcmxkIQo=" }
```

AWS AppSync converts it to a string containing the base64 representation of the value:

```
"SGVsbG8sIFdvcmxkIQo="
```

Note that the binary data is encoded in the base64 encoding scheme as specified in RFC 4648 and RFC 2045.

Binary set type BS

A set of binary values. A DynamoDB Binary Set value is returned as a list of strings containing the base64 representation of the values.

For example, if DynamoDB returned the following DynamoDB Binary Set value:

```
{ "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ] }
```

AWS AppSync converts it to a list of strings containing the base64 representation of the values:

```
[ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ]
```

Note that the binary data is encoded in the base64 encoding scheme as specified in RFC 4648 and RFC 2045.

Boolean type BOOL

A Boolean value. A DynamoDB Boolean value is returned as a Boolean.

For example, if DynamoDB returned the following DynamoDB Boolean value:

```
{ "BOOL" : true }
```

AWS AppSync converts it to a Boolean:

```
true
```

List type L

A list of any other supported DynamoDB value. A DynamoDB List value is returned as a list of values, where each inner value is also converted.

For example, if DynamoDB returned the following DynamoDB List value:

```
{ "L" : [
      { "S"  : "A string value" },
      { "N"  : 1 },
      { "SS" : [ "Another string value", "Even more string values!" ] }
   ]
```

```
}
```

AWS AppSync converts it to a list of converted values:

```
[ "A string value", 1, [ "Another string value", "Even more string values!" ] ]
```

Map type `M`

A key/value collection of any other supported DynamoDB value. A DynamoDB Map value is returned as a JSON object, where each key/value is also converted.

For example, if DynamoDB returned the following DynamoDB Map value:

```
{ "M" : {
      "someString" : { "S"  : "A string value" },
      "someNumber" : { "N"  : 1 },
      "stringSet"  : { "SS" : [ "Another string value", "Even more string
 values!" ] }
   }
}
```

AWS AppSync converts it to a JSON object:

```
{
   "someString" : "A string value",
   "someNumber" : 1,
   "stringSet"  : [ "Another string value", "Even more string values!" ]
}
```

Null type `NULL`

A null value.

For example, if DynamoDB returned the following DynamoDB Null value:

```
{ "NULL" : null }
```

AWS AppSync converts it to a null:

```
null
```

Filters

> **Note**
>
> We recommend using the DynamoDB built-in module to generate your request. For more information, see Amazon DynamoDB built-in module.

When querying objects in DynamoDB using the `Query` and `Scan` operations, you can optionally specify a `filter` that evaluates the results and returns only the desired values.

The filter property of a `Query` or `Scan` request has the following structure:

```
type DynamoDBExpression = {
  expression: string;
  expressionNames?: { [key: string]: string};
  expressionValues?: { [key: string]: any};
};
```

The fields are defined as follows:

`expression`

The query expression. For more information about how to write filter expressions, see the DynamoDB QueryFilter and DynamoDB ScanFilter documentation. This field must be specified.

`expressionNames`

The substitutions for expression attribute *name* placeholders, in the form of key-value pairs. The key corresponds to a name placeholder used in the `expression`. The value must be a string that corresponds to the attribute name of the item in DynamoDB. This field is optional, and should only be populated with substitutions for expression attribute name placeholders used in the `expression`.

`expressionValues`

The substitutions for expression attribute *value* placeholders, in the form of key-value pairs. The key corresponds to a value placeholder used in the `expression`, and the value must be a typed value. For more information about how to specify a “typed value”, see Type system (request mapping). This must be specified. This field is optional, and should only be populated with substitutions for expression attribute value placeholders used in the `expression`.

Example

The following example is a filter section for a request, where entries retrieved from DynamoDB are only returned if the title starts with the `title` argument.

Here we use the `util.transform.toDynamoDBFilterExpression` to automatically create a filter from an object:

```
const filter = util.transform.toDynamoDBFilterExpression({
  title: { beginsWith: 'far away' },
});

const request = {};
request.filter = JSON.parse(filter);
```

This generates the following filter:

```
{
  "filter": {
    "expression": "(begins_with(#title,:title_beginsWith))",
    "expressionNames": { "#title": "title" },
    "expressionValues": {
      ":title_beginsWith": { "S": "far away" }
    }
  }
}
```

Condition expressions

When you mutate objects in DynamoDB by using the `PutItem`, `UpdateItem`, and `DeleteItem` DynamoDB operations, you can optionally specify a condition expression that controls whether the request should succeed or not, based on the state of the object already in DynamoDB before the operation is performed.

While you can construct requests manually, we recommend using the DynamoDB built-in module to generate accurate and efficient requests. In the following examples, we use the built-in module to generate requests with conditions.

The AWS AppSync DynamoDB function allows a condition expression to be specified in `PutItem`, `UpdateItem`, and `DeleteItem` request objects, and also a strategy to follow if the condition fails and the object was not updated.

Example 1

The following `PutItem` request object doesn't have a condition expression. As a result, it puts an item in DynamoDB even if an item with the same key already exists, which overwrites the existing item.

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

export const onPublish = {
  request(ctx) {
    const {id, payload: item} = ctx.events[0]
    return ddb.put({ key: { id }, item })
  },
  response: (ctx) => ctx.events
}
```

Example 2

The following `PutItem` object does have a condition expression that allows the operation to succeed only if an item with the same key does *not* exist in DynamoDB.

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

export const onPublish = {
  request(ctx) {
    const {id, payload: item} = ctx.events[0]
    return ddb.put({
      key: { id },
      item,
      condition: {id: {attributeExists: false}}
    })
  },
  response: (ctx) => ctx.events
}
```

For more information about DynamoDB conditions expressions, see the DynamoDB ConditionExpressions documentation .

Specifying a condition

The `PutItem`, `UpdateItem`, and `DeleteItem` request objects all allow an optional `condition` section to be specified. If omitted, no condition check is made. If specified, the condition must be true for the operation to succeed.

The built-in module functions create a `condition` object that has the following structure.

```
type ConditionCheckExpression = {
  expression: string;
  expressionNames?: { [key: string]: string};
  expressionValues?: { [key: string]: any};
};
```

The following fields specify the condition:

`expression`

The update expression itself. For more information about how to write condition expressions, see the DynamoDB ConditionExpressions documentation . This field must be specified.

`expressionNames`

The substitutions for expression attribute name placeholders, in the form of key-value pairs. The key corresponds to a name placeholder used in the *expression*, and the value must be a string corresponding to the attribute name of the item in DynamoDB. This field is optional, and should only be populated with substitutions for expression attribute name placeholders used in the *expression*.

`expressionValues`

The substitutions for expression attribute value placeholders, in the form of key-value pairs. The key corresponds to a value placeholder used in the expression, and the value must be a typed value. For more information about how to specify a “typed value”, see Type system (request mapping). This must be specified. This field is optional, and should only be populated with substitutions for expression attribute value placeholders used in the expression.

Transaction condition expressions

Transaction condition expressions are available in requests of all four types of operations in `TransactWriteItems`, namely, `PutItem`, `DeleteItem`, `UpdateItem`, and `ConditionCheck`.

For `PutItem`, `DeleteItem`, and `UpdateItem`, the transaction condition expression is optional. For `ConditionCheck`, the transaction condition expression is required.

Example 1

The following transactional `DeleteItem` function request handler does not have a condition expression. As a result, it deletes the item in DynamoDB.

```
export const onPublish = {
  request(ctx) {
    const table = "events"
    return ddb.transactWrite({
      items: ctx.events.map(({ payload }) => ({
        deleteItem: { table, key: { id: payload.id } }
      }))
    })
  },
  response: (ctx) => ctx.events
}
```

Example 2

The following transactional `DeleteItem` function request handler does have a transaction condition expression that allows the operation succeed only if the author of that post equals a certain name.

```
export const onPublish = {
  request(ctx) {
    return ddb.remove({
      items: ctx.events.map(({ payload }) => ({
        deleteItem: {
          table: 'events',
          key: { id: payload.id },
          condition: { owner: { eq: payload.owner } }
        }
      }))
    })
  },
  response: (ctx) => ctx.events
}
```

If the condition check fails, it will cause `TransactionCanceledException` and the error detail will be returned in `ctx.result.cancellationReasons`.

Projections

When reading objects in DynamoDB using the `GetItem`, `Scan`, `Query`, `BatchGetItem`, and `TransactGetItems` operations, you can optionally specify a projection that identifies the attributes that you want. The projection property has the following structure, which is similar to filters:

```
type DynamoDBExpression = {
  expression: string;
  expressionNames?: { [key: string]: string}
};
```

The fields are defined as follows:

`expression`

The projection expression, which is a string. To retrieve a single attribute, specify its name. For multiple attributes, the names must be comma-separated values. For more information on writing projection expressions, see the DynamoDB projection expressions documentation. This field is required.

`expressionNames`

The substitutions for expression attribute *name* placeholders in the form of key-value pairs. The key corresponds to a name placeholder used in the `expression`. The value must be a string that corresponds to the attribute name of the item in DynamoDB. This field is optional and should only be populated with substitutions for expression attribute name placeholders used in the `expression`. For more information about `expressionNames`, see the DynamoDB documentation.

Example 1

The following example is a projection section for a JavaScript function in which only the attributes `author` and `id` are returned from DynamoDB.

```
projection : {
```

```
    expression : "#author, id",
    expressionNames : {
        "#author" : "author"
    }
}
```

Example 2

The following example demonstrates that when you use the built-in DynamoDB module, you can simply pass an array for your projection.

```
export const onPublish = {
  request(ctx) {
    return ddb.batchGet({
      tables: {
        users: {
          keys: ctx.events.map(e => ({id: e.payload.id})),
          projection: ['id', 'name', 'email', 'nested.field']
        }
      }
    })
  },
  response: (ctx) => ctx.events
}
```

AWS AppSync JavaScript function reference for Amazon OpenSearch Service

The AWS AppSync integration for Amazon OpenSearch Service enables you to store and retrieve data in existing OpenSearch Service domains in your account. This handler works by allowing you to create OpenSearch Service requests, and then map the OpenSearch Service response back to your application. This section describes the function request and response handlers for the supported OpenSearch Service operations.

Request

Most OpenSearch Service request objects have a common structure where just a few pieces change. The following example runs a search against an OpenSearch Service domain, where documents are of type `post` and are indexed under `id`. The search parameters are defined in the `body` section,

with many of the common query clauses being defined in the `query` field. This example will search for documents containing `"Nadia"`, or `"Bailey"`, or both, in the `author` field of a document:

```
export const onPublish = {
  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' } },
              ],
            },
          },
        },
      },
    };
  }
}
```

Response

As with other data sources, OpenSearch Service sends a response to AWS AppSync that needs to be processed. .

Most applications are looking for the `_source` field from an OpenSearch Service response. Because you can do searches to return either an individual document or a list of documents, there are two common response patterns used in OpenSearch Service.

List of Results

```
export const onPublish = {
  response(ctx) {
    const entries = [];
```

```
    for (const entry of ctx.result.hits.hits) {
      entries.push(entry['_source']);
    }
  }
}
```

Individual Item

```
export const onPublish = {
  response(ctx) {
    const result =  ctx.result['_source']
  }
}
```

`operation` field

> **Note**
>
> This applies only to the Request handler.

HTTP method or verb (GET, POST, PUT, HEAD or DELETE) that AWS AppSync sends to the OpenSearch Service domain. Both the key and the value must be a string.

```
"operation" : "PUT"
```

`path` field

> **Note**
>
> This applies only to the Request handler.

The search path for an OpenSearch Service request from AWS AppSync. This forms a URL for the operation's HTTP verb. Both the key and the value must be strings.

```
"path" : "/indexname/type"

"path" : "/indexname/type/_search"
```

When the request handler is evaluated, this path is sent as part of the HTTP request, including the OpenSearch Service domain. For example, the previous example might translate to:

```
GET https://opensearch-domain-name.REGION.es.amazonaws.com/indexname/type/_search
```

`params` field

> **Note**
>
> This applies only to the Request handler.

Used to specify what action your search performs, most commonly by setting the **query** value inside of the **body**. However, there are several other capabilities that can be configured, such as the formatting of responses.

- **headers**

  The header information, as key-value pairs. Both the key and the value must be strings. For example:

  ```
  "headers" : {
      "Content-Type" : "application/json"
  }
  ```

  > **Note**
  >
  > AWS AppSync currently supports only JSON as a `Content-Type`.

- **queryString**

  Key-value pairs that specify common options, such as code formatting for JSON responses. Both the key and the value must be a string. For example, if you want to get pretty-formatted JSON, you would use:

  ```
  "queryString" : {
      "pretty" : "true"
  }
  ```

- **body**

  This is the main part of your request, allowing AWS AppSync to craft a well-formed search request to your OpenSearch Service domain. The key must be a string comprised of an object. A couple of demonstrations are shown below.

Example 1

Return all documents with a city matching “seattle”:

```
export const onSubscribe = {
  request(ctx) {
    return {
      operation: 'GET',
      path: '/id/post/_search',
      params: {
        headers: {},
        queryString: {},
        body: { from: 0, size: 50, query: { match: { city: 'seattle' } } },
      },
    };
  }
}
```

Example 2

Return all documents matching “washington” as the city or the state:

```
export const onSubscribe = {
  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'] },
          },
```

```
        },
      },
    };
  }
}
```

AWS AppSync JavaScript function reference for Lambda

You can use AWS AppSync integration for AWS Lambda to invoke Lambda functions located in your account. You can shape your request payloads and the response from your Lambda functions before returning them to your clients. You can also specify the type of operation to perform in your request object. This section describes the requests for the supported Lambda operations.

Request object

The Lambda request object handles fields related to your Lambda function:

```
export type LambdaRequest = {
  operation: 'Invoke';
  invocationType?: 'RequestResponse' | 'Event';
  payload: unknown;
};
```

The following example uses an `invoke` operation with its payload data being a field, along with its arguments from the context:

```
export const onPublish = {
  request(ctx) {
    return {
      operation: 'Invoke',
      payload: { field: 'getPost', arguments: ctx.args },
    };
  }
}
```

Operation

When doing an `Invoke`, the resolved request matches the input payload of the *Lambda* function. The following example modifies the previous example:

```
export const onPublish = {
```

```
  request(ctx) {
    return {
      operation: 'Invoke',
      payload: ctx // send the entire context to the Lambda function
    };
  }
}
```

Payload

The `payload` field is a container used to pass any data to the Lambda function. The `payload` field is optional.

Invocation type

The Lambda data source allows you to define two invocation types: `RequestResponse` and `Event`. The invocation types are synonymous with the invocation types defined in the Lambda API. The `RequestResponse` invocation type lets AWS AppSync call your Lambda function synchronously to wait for a response. The `Event` invocation allows you to invoke your Lambda function asynchronously. For more information on how Lambda handles `Event` invocation type requests, see Asynchronous invocation. The `invocationType` field is optional. If this field is not included in the request, AWS AppSync will default to the `RequestResponse` invocation type.

For any `invocationType` field, the resolved request matches the input payload of the Lambda function. The following example modifies the previous example:

```
export const onPublish = {
  request(ctx) {
     return {
       operation:  'Invoke',
       invocationType:  'Event',
       payload: ctx
    };
  }
}
```

Response object

As with other data sources, your Lambda function sends a response to AWS AppSync that must be processed. The result of the Lambda function is contained in the `context` result property (`context.result`).

If the shape of your Lambda function response matches the expected output, you can forward the response using the following function response handler:

```
export const onPublish = {
  respone(ctx) {
    console.log(`the response: ${ctx.result}`)
    return ctx.events
  }
}
```

There are no required fields or shape restrictions that apply to the response object.

AWS AppSync JavaScript function reference for EventBridge data source

The AWS AppSync integration for Amazon EventBridge data source allows you to send custom events to the EventBridge bus.

Request

The request handler allows you to send multiple custom events to an EventBridge event bus:

```
export const onPublish = {
  request(ctx) {
    return {
      "operation" : "PutEvents",
      "events" : ctx.events.map(e => ({
        source: ctx.info.channel.path,
        detail: {payload: e.payload},
        detailType: ctx.info.channelNamespace.name,
      }))
    }
  }
}
```

An EventBridge `PutEvents` request has the following type definition:

```
type PutEventsRequest = {
  operation: 'PutEvents'
```

```
  events: {
    source: string
    detail: { [key: string]: any }
    detailType: string
    resources?: string[]
    time?: string // RFC3339 Timestamp format
  }[]
}
```

Response

If the `PutEvents` operation is successful, the response from EventBridge is included in the `ctx.result`:

```
export function response(ctx) {
  if(ctx.error)
    util.error(ctx.error.message, ctx.error.type, ctx.result)
  else
    return ctx.result
}
```

Errors that occur while performing `PutEvents` operations such as `InternalExceptions` or `Timeouts` will appear in `ctx.error`. For a list of EventBridge's common errors, see the EventBridge common error reference.

The `result` will have the following type definition:

```
type PutEventsResult = {
  Entries: {
    ErrorCode: string
    ErrorMessage: string
    EventId: string
  }[]
  FailedEntryCount: number
}
```

- **Entries**

  The ingested event results, both successful and unsuccessful. If the ingestion was successful, the entry has the `EventID` in it. Otherwise, you can use the `ErrorCode` and `ErrorMessage` to identify the problem with the entry.

For each record, the index of the response element is the same as the index in the request array.

- **FailedEntryCount**

  The number of failed entries. This value is represented as an integer.

For more information about the response of `PutEvents`, see PutEvents.

Example sample response 1

The following example is a `PutEvents` operation with two successful events:

```
{
    "Entries" : [
        {
            "EventId": "11710aed-b79e-4468-a20b-bb3c0c3b4860"
        },
        {
            "EventId": "d804d26a-88db-4b66-9eaf-9a11c708ae82"
        }
    ],
    "FailedEntryCount" : 0
}
```

Example sample response 2

The following example is a `PutEvents` operation with three events, two successes and one fail:

```
{
    "Entries" : [
        {
            "EventId": "11710aed-b79e-4468-a20b-bb3c0c3b4860"
        },
        {
            "EventId": "d804d26a-88db-4b66-9eaf-9a11c708ae82"
        },
        {
            "ErrorCode" : "SampleErrorCode",
            "ErrorMessage" : "Sample Error Message"
        }
    ],
    "FailedEntryCount" : 1
```

```
}
```

PutEvents fields

PutEvents contains the following mapping template fields:

- **Version**

  Common to all request mapping templates, the version field defines the version that the template uses. This field is required. The value 2018-05-29 is the only version supported for the EventBridge mapping templates.

- **Operation**

  The only supported operation is PutEvents. This operation allows you to add custom events to your event bus.

- **Events**

  An array of events that will be added to the event bus. This array should have an allocation of 1 - 10 items.

  The Event object has the following fields:

  - "source": A string that defines the source of the event.
  - "detail": A JSON object that you can use to attach information about the event. This field can be an empty map ({ }).
  - "detailType: A string that identifies the type of event.
  - "resources": A JSON array of strings that identifies resources involved in the event. This field can be an empty array.
  - "time": The event timestamp provided as a string. This should follow the RFC3339 timestamp format.

The following are examples of valid Event objects:

Example 1

```
{
    "source" : "source1",
    "detail" : {
```

```
        "key1" : [1,2,3,4],
        "key2" : "strval"
    },
    "detailType" : "sampleDetailType",
    "resources" : ["Resouce1", "Resource2"],
    "time" : "2022-01-10T05:00:10Z"
}
```

Example 2

```
{
    "source" : "source1",
    "detail" : {},
    "detailType" : "sampleDetailType"
}
```

Example 3

```
{
    "source" : "source1",
    "detail" : {
        "key1" : 1200
    },
    "detailType" : "sampleDetailType",
    "resources" : []
}
```

AWS AppSync JavaScript function reference for HTTP

The AWS AppSync HTTP functions enable you to send requests from AWS AppSync to any HTTP endpoint, and responses from your HTTP endpoint back to AWS AppSync. With your request handler, you can provide hints to AWS AppSync about the nature of the operation to be invoked. This section describes the different configurations for the supported HTTP resolver.

Request

```
type HTTPRequest = {
  method: 'PUT' | 'POST' | 'GET' | 'DELETE' | 'PATCH';
  params?: {
    query?: { [key: string]: any };
```

```
    headers?: { [key: string]: string };
    body?: any;
  };
  resourcePath: string;
};
```

The following is an example of an HTTP POST request, with a `text/plain` body:

```
export const onPublish = {
  request(ctx) {
    return {
      resourcePath: '/',
      method: 'POST',
      params: {
        headers: { 'Content-Type': 'text/plain' },
        body: 'this is an example of text body',
      }
    };
  }
}
```

Method

HTTP method or verb (GET, POST, PUT, PATCH, or DELETE) that AWS AppSync sends to the HTTP endpoint.

```
"method": "PUT"
```

ResourcePath

The resource path that you want to access. Along with the endpoint in the HTTP data source, the resource path forms the URL that the AWS AppSync service makes a request to.

```
"resourcePath": "/v1/users"
```

When the request is evaluated, this path is sent as part of the HTTP request, including the HTTP endpoint. For example, the previous example might translate to the following:

```
PUT <endpoint>/v1/users
```

Params fields

headers

The header information, as key-value pairs. Both the key and the value must be strings.

For example:

```
"headers" : {
    "Content-Type" : "application/json"
}
```

Currently supported `Content-Type` headers are:

```
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
```

You can't set the following HTTP headers:

```
HOST
CONNECTION
USER-AGENT
EXPECTATION
TRANSFER_ENCODING
CONTENT_LENGTH
```

query

Key-value pairs that specify common options, such as code formatting for JSON responses. Both the key and the value must be a string. The following example shows how you can send a query string as `?type=json`:

```
"query" : {
    "type" : "json"
}
```

body

The body contains the HTTP request body that you choose to set. The request body is always a UTF-8 encoded string unless the content type specifies the charset.

```
"body":"body string"
```

Response

The response of the request is available in `ctx.result`. If the request results in an error, the error is available in `ctx.error`. You can check the status of the response in `ctx.result.statusCode`, and get the body returned in the response in `ctx.result.body`.

AWS AppSync JavaScript function reference for Amazon RDS

The AWS AppSync RDS function enables you to send SQL queries to an Amazon Aurora cluster database using the RDS Data API and get back the result of these queries. You can write SQL statements that are sent to the Data API by using AWS AppSync's `rds` module `sql`-tagged template or by using the `rds` module's `select`, `insert`, `update`, and `remove` helper functions. AWS AppSync utilizes the RDS Data Service's `ExecuteStatement` action to run SQL statements against the database.

SQL tagged template

AWS AppSync's `sql` tagged template enables you to create a static statement that can receive dynamic values at runtime by using template expressions. AWS AppSync builds a variable map from the expression values to construct a `SqlParameterized` query that is sent to the Amazon Aurora Serverless Data API. With this method, it isn't possible for dynamic values passed at run time to modify the original statement, which could cause unintented execution. All dynamic values are passed as parameters, can't modify the original statement, and aren't executed by the database. This makes your query less vulnerable to SQL injection attacks.

> **Note**
>
> In all cases, when writing SQL statements, you should follow security guidelines to properly handle data that you receive as input.

> **Note**
>
> The `sql` tagged template only supports passing variable values. You can't use an expression to dynamically specify the column or table names. However, you can use utility functions to build dynamic statements.

Filtering Database Results Securely with Dynamic Channel Paths

When building AWS AppSync applications, you often need to filter database queries based on dynamic values. This pattern shows how to safely incorporate run-time values into your SQL queries while maintaining security. In the following example, we create a query that filters based on the value of channel path that is set dynamically at run time. The value can easily be added to the statement using the tag expression.

```
import { sql, createMySQLStatement as mysql } from '@aws-appsync/utils/rds';

    export const onPublish = {
      request(ctx) {
        const query = sql`
    SELECT * FROM table
    WHERE column = ${ctx.info.channel.path}`;
        return mysql(query);
      }
    }
```

The database engine automatically protects against SQL injection attacks by sanitizing all values passed through the variable map.

Creating statements

Handlers can interact with MySQL and PostgreSQL databases. Use `createMySQLStatement` and `createPgStatement` respectively to build statements. For example, `createMySQLStatement` can create a MySQL query. These functions accept up to two statements, useful when a request should retrieve results immediately. With MySQL, you can do the following:

```
import { sql, createMySQLStatement } from '@aws-appsync/utils/rds';

export const onSubscribe = {
```

```
  request(ctx) {
    const { id, text } = ctx.events[0].payload;
    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` and `createMySQLStatement` does not escape or quote statements built with the `sql` tagged template.

Retrieving data

The result of your executed SQL statement is available in your response handler in the `context.result` object. The result is a JSON string with the response elements from the `ExecuteStatement` action. When parsed, the result has the following shape:

```
type SQLStatementResults = {
    sqlStatementResults: {
        records: any[];
        columnMetadata: any[];
        numberOfRecordsUpdated: number;
        generatedFields?: any[]
    }[]
}
```

The following example demonstrates how you can use the `toJsonObject` utility to transform the result into a list of JSON objects representing the returned rows.

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

export const onSubscribe = {
  response(ctx) {
    const { error, result } = ctx;
    if (error) {
      return util.error(
        error.message,
        error.type,
```

```
        result
      )
    }
    const result =  toJsonObject(result)[1][0]
  }
}
```

Note that `toJsonObject` returns an array of statement results. If you provided one statement, the array length is 1. If you provided two statements, the array length is 2. Each result in the array contains `0` or more rows. `toJsonObject` returns `null` if the result value is invalid or unexpected.

Utility functions

You can use the AWS AppSync RDS module's utility helpers to interact with your database. To learn more, see Amazon RDS module functions.

AWS AppSync JavaScript function reference for Amazon Bedrock

You can use AWS AppSync functions to invoke models on Amazon Bedrock in your AWS account. You can shape your request payloads and the response from your model invocations functions before returning them to your clients. You can use the Amazon Bedrock runtime's `InvokeModel` API or the `Converse` API. This section describes the requests for the supported Amazon Bedrock operations.

> **Note**
>
> AWS AppSync only supports synchronous invocations that complete within 10 seconds. It is not possible to call Amazon Bedrock's stream APIs. AWS AppSync only supports invoking foundation models and inference profiles in the same region as the AWS AppSync API.

Request object

The `InvokeModel` request object allows you to interact with Amazon Bedrock's `InvokeModel` API.

```
type BedrockInvokeModelRequest = {
  operation: 'InvokeModel';
```

```
  modelId: string;
  body: any;
  guardrailIdentifier?: string;
  guardrailVersion?: string;
  guardrailTrace?: string;
}
```

The `Converse` request object allows you to interact with Amazon Bedrock's `Converse` API.

```
type BedrockConverseRequest = {
  operation: 'Converse';
  modelId: string;
  messages: BedrockMessage[];
  additionalModelRequestFields?: any;
  additionalModelResponseFieldPaths?: string[];
  guardrailConfig?: BedrockGuardrailConfig;
  inferenceConfig?: BedrockInferenceConfig;
  promptVariables?: { [key: string]: BedrockPromptVariableValues }[];
  system?: BedrockSystemContent[];
  toolConfig?: BedrockToolConfig;
}
```

See the Type reference section later in this topic for more details.

From your functions and resolvers, you can build your request objects directly or use the helper functions from @aws-appsync/utils/ai to create the request. When specifying the model Id (modelId) in your requests, you can use the model Id or the model ARN.

The following example uses the `invokeModel` function to summarize text using Amazon Titan Text G1 - Lite (amazon.titan-text-lite-v1). A configured guardrail is used to identify and block or filter unwanted content in the prompt flow. Learn more about Amazon Bedrock Guardrails in the *Amazon Bedrock User Guide*.

> **Important**
>
> You are responsible for secure application development and preventing vulnerabilities, such as prompt injection. To learn more, see Prompt injection security in the *Amazon Bedrock User Guide*.

```
import { invokeModel } from '@aws-appsync/utils/ai'
```

```
export const onPublish = {
  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 const onProcessResult = {
  response(ctx) {
    return ctx.result.results[0].outputText
  }
}
```

The following example uses the `converse` function with a cross-region inference profile (us.anthropic.claude-3-5-haiku-20241022-v1:0). Learn more about Amazon Bedrock's Prerequisites for inference profiles in the *Amazon Bedrock User Guide*

Reminder: You are responsible for secure application development and preventing vulnerabilities, such as prompt injection.

```
import { converse } from '@aws-appsync/utils/ai'

export const onPublish = {
  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 const onProcessResult = {
  response(ctx) {
    return ctx.result.output.message.content[0].text
  }
}
```

The following example uses `converse` to create a structured response. Note that we use environment variables for our DB schema reference and we configure a guardrail to help prevent attacks.

```
import { converse } from '@aws-appsync/utils/ai'

export const onPublish = {
  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' },
    })
  },
    response(ctx) {
    const result = toolReponse(ctx.result)
    return []
  }
}

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
}
```

Given the schema:

```
type SQLResult {
    sql: String
    parameters: AWSJSON
}

type Query {
    db(text: String!): SQLResult
}
```

and the query:

```
query db($text: String!) {
  db(text: $text) {
    parameters
    sql
  }
}
```

With the following parameters:

```
{
  "text":"What is my top selling product?"
}
```

The following response is returned:

```
{
  "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
    }
  }
}
```

However, with this request:

```
{
  "text":"give me a query to retrieve sensitive information"
}
```

The following response is returned:

```
{
  "data": {
    "db": {
      "parameters": null,
      "sql": "SELECT null; -- I cannot and will not assist with retrieving sensitive
 private information"
    }
  }
}
```

To learn more about configuring Amazon Bedrock Guardrails, see Stop harmful content in models using Amazon Bedrock Guardrails in the *Amazon Bedrock User Guide*.

Response object

The response from your Amazon Bedrock runtime invocation is contained in the context's result property (`ctx.result`). The response matches the shape specified by Amazon Bedrock's APIs. See the Amazon Bedrock User Guide for more information about the expected shape of invocation results.

```
export const onPublish = {
  response(ctx) {
    return ctx.result
  }
```

```
}
```

Long running invocations

Many organizations currently use AWS AppSync as an AI gateway to build generative AI applications that are powered by foundation models on Amazon Bedrock. Customers use AWS AppSync subscriptions, powered by WebSockets, to return progressive updates from long-running model invocations. This allows them to implement asynchronous patterns.

The following diagram demonstrates how you can implement this pattern. In the diagram, the following steps occur.

1. Your client starts a subscription, which sets up a WebSocket, and makes a request to AWS AppSync to trigger a Generative AI invocation.
2. AWS AppSync calls your AWS Lambda function in Event mode and immediately returns a response to the client.
3. Your Lambda function invokes the model on Amazon Bedrock. The Lambda function can use a synchronous API, such as `InvokeModel`, or a stream API, such as `InvokeModelWithResponseStream`, to get progressive updates.
4. As updates are received, or when the invocation completes, the Lambda function sends updates via mutations to your AWS AppSync API which triggers subscriptions.
5. The subscription events are sent in real-time and received by your client over the WebSocket.

Type reference

```
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?: BedrockGuardrailTraceAsssessment;
};

export type BedrockGuardrailTraceAsssessment = {
  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;
};
```

Document History for AWS AppSync Events

The following table describes the documentation for this release of AWS AppSync Events.

- **API version: 1.1**
- **Latest documentation update:** April 24, 2025

Change	Description	Date
AWS AppSync Events API integrates with AWS data sources	AWS AppSync Events supports multiple AWS data sources that you can interact with from your channel namespace handlers. To learn more, see Working with data sources for AWS AppSync Event APIs.	April 24, 2025
AWS AppSync Events supports publishing via WebSocket	AWS AppSync Events allows you to publish events via your API's WebSocket endpoint after you connect to it. To learn more, see Publish events via WebSocket.	March 13, 2025
AWS AppSync Events release	This release introduces AWS AppSync Events.	October 30, 2024