SDK 버전 2용 개발자 가이드

AWS SDK for JavaScript

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

AWS SDK for JavaScript: SDK 버전 2용 개발자 가이드

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

Amazon의 상표 및 트레이드 드레스는 Amazon 외 제품 또는 서비스와 함께, Amazon 브랜드 이미지를 떨어뜨리거나 고객에게 혼동을 일으킬 수 있는 방식으로 사용할 수 없습니다. Amazon이 소유하지 않은 기타 모든 상표는 Amazon과 제휴 관계이거나 관련이 있거나 후원 관계와 관계없이 해당 소유자의 자산입니다.

Table of Contents

.... ix
AWS SDK for JavaScript란 무엇인가요? 1
SDK 메이저 버전에 대한 유지 관리 및 지원 1
Node.js에서 SDK 사용 2
AWS Amplify에서 SDK 사용 2
웹 브라우저에서 SDK 사용 2
일반 사용 사례 2
예제 정보 3
시작하기 4
브라우저 스크립트에서 시작하기 4
시나리오 4
1단계: Amazon Cognito 자격 증명 풀 생성 5
2단계: 생성된 IAM 역할에 정책 추가 6
3단계: HTML 페이지 만들기 7
4단계: 브라우저 스크립트 작성 8
5단계: 샘플 실행 9
전체 샘플 10
가능한 개선 사항 11
Node.js에서 시작하기 11
시나리오 12
사전 필수 작업 12
1단계: SDK 및 종속성 설치 12
2단계: 자격 증명 구성 13
3단계: 프로젝트용 패키지 JSON 생성 14
4단계: Node.js 작성 14
5단계: 샘플 실행 16
SDK for JavaScript 설정 17
사전 조건 17
AWS Node.js 환경 설정 17
지원되는 웹 브라우저 18
SDK 설치 19
Bower를 사용한 설치 20
SDK 로드 20
버전 1에서 업그레이드 21

입력/출력 시 Base64 및 타임스탬프의 자동 변환 ... 21
response.data.RequestId를 response.requestId로 이동 ... 22
노출된 래퍼 요소 ... 22
삭제된 클라이언트 속성 ... 27
SDK for JavaScript 구성 ... 28
글로벌 구성 객체 사용하기 ... 28
글로벌 구성 설정 ... 29
서비스별 구성 설정 ... 30
변경 불가능한 구성 데이터 ... 31
AWS 리전 설정 ... 31
클라이언트 클래스 생성자에서 ... 31
글로벌 구성 객체 사용하기 ... 32
환경 변수 사용 ... 32
공유 구성 파일 사용 ... 32
리전 설정을 위한 우선 순위 ... 32
사용자 지정 엔드포인트 지정 ... 33
엔드포인트 문자열 형식 ... 33
ap-northeast-3 리전의 엔드포인트 ... 33
MediaConvert용 엔드포인트 ... 33
AWS를 사용한 SDK 인증 ... 34
AWS 액세스 포털 세션 시작 ... 35
세부 인증 정보 ... 36
자격 증명 설정 ... 36
인증 자격 증명에 대한 모범 사례 ... 37
Node.js에서 자격 증명 설정 ... 37
웹 브라우저에서 자격 증명 설정 ... 42
API 버전 잠금 ... 52
API 버전 가져오기 ... 52
Node.js 고려 사항 ... 52
기본 제공 Node.js 모듈 사용 ... 53
NPM 패키지 사용 ... 53
Node.js에서 maxSockets 구성 ... 54
Node.js에서 연결 유지를 이용해 연결 재사용 ... 55
Node.js용 프록시 구성 ... 56
Node.js에서 인증서 번들 등록 ... 57
브라우저 스크립트 고려 사항 ... 57

브라우저용 SDK 빌드 57
CORS(Cross-Origin Resource Sharing) 60
Webpack과 번들링 64
Webpack 설치 64
Webpack 구성 65
Webpack 실행 66
Webpack 번들 사용 67
개별 서비스 가져오기 67
Node.js에 대한 번들링 68
서비스 작업 70
서비스 객체 생성 및 호출 71
개별 서비스 필요 72
서비스 객체 생성 73
서비스 객체의 API 버전 잠금 73
서비스 객체 파라미터 지정 74
AWS SDK for JavaScript 호출 로깅 74
타사 로거 사용 75
비동기식 서비스 호출 75
비동기식 호출 관리 76
콜백 함수 사용 77
요청 객체 이벤트 리스너 사용 78
비동기/대기 사용 84
Promises 사용 84
응답 객체 사용 87
응답 객체에서 반환된 데이터에 액세스 87
반환된 데이터를 통해 페이징 88
응답 객체에서 오류 정보에 액세스 89
발신 요청 객체에 액세스 89
JSON 작업 89
서비스 객체 파라미터로서 JSON 90
JSON으로 데이터 반환 91
Retries 92
지수 백오프 기반 재시도 동작 92
SDK for JavaScript 코드 예제 95
Amazon CloudWatch 예제 95
Amazon CloudWatch 경보 생성 96

- Amazon CloudWatch 경보 작업 사용 ... 100
- Amazon CloudWatch에서 지표 가져오기 ... 104
- Amazon CloudWatch Events에 이벤트 전송 ... 107
- Amazon CloudWatch Logs 구독 필터 사용 ... 112
Amazon DynamoDB 예제 ... 117
- DynamoDB에서 테이블 생성 및 사용 ... 117
- DynamoDB에서 단일 항목 읽기 및 쓰기 ... 122
- DynamoDB에서 배치로 항목 읽기 및 쓰기 ... 126
- DynamoDB 테이블 쿼리 및 스캔 ... 129
- DynamoDB 문서 클라이언트 사용 ... 132
Amazon EC2 예제 ... 138
- Amazon EC2 인스턴스 생성 ... 139
- Amazon EC2 인스턴스 관리 ... 142
- Amazon EC2 키 페어로 작업 ... 148
- Amazon EC2에서 리전 및 가용 영역 사용 ... 151
- Amazon EC2의 보안 그룹 작업 ... 153
- Amazon EC2에서 탄력적 IP 주소 사용 ... 158
MediaConvert 예제 ... 162
- 작업 생성 및 관리 ... 162
- 작업 템플릿 사용 ... 169
AWS IAM 예제 ... 178
- IAM 사용자 관리 ... 179
- IAM 정책 작업 ... 183
- IAM 액세스 키 관리 ... 189
- IAM 서버 인증서 작업 ... 194
- IAM 계정 별칭 관리 ... 198
Amazon Kinesis 예제 ... 201
- Amazon Kinesis를 사용하여 웹 페이지 스크롤 진행 상황 캡처 ... 202
Amazon S3 예제 ... 209
- Amazon S3 브라우저 예제 ... 210
- Amazon S3 Node.js 예제 ... 238
Amazon SES 예제 ... 258
- 자격 증명 관리 ... 259
- 이메일 템플릿 사용 ... 264
- Amazon SES로 이메일 전송 ... 269
- IP 주소 필터 사용 ... 275

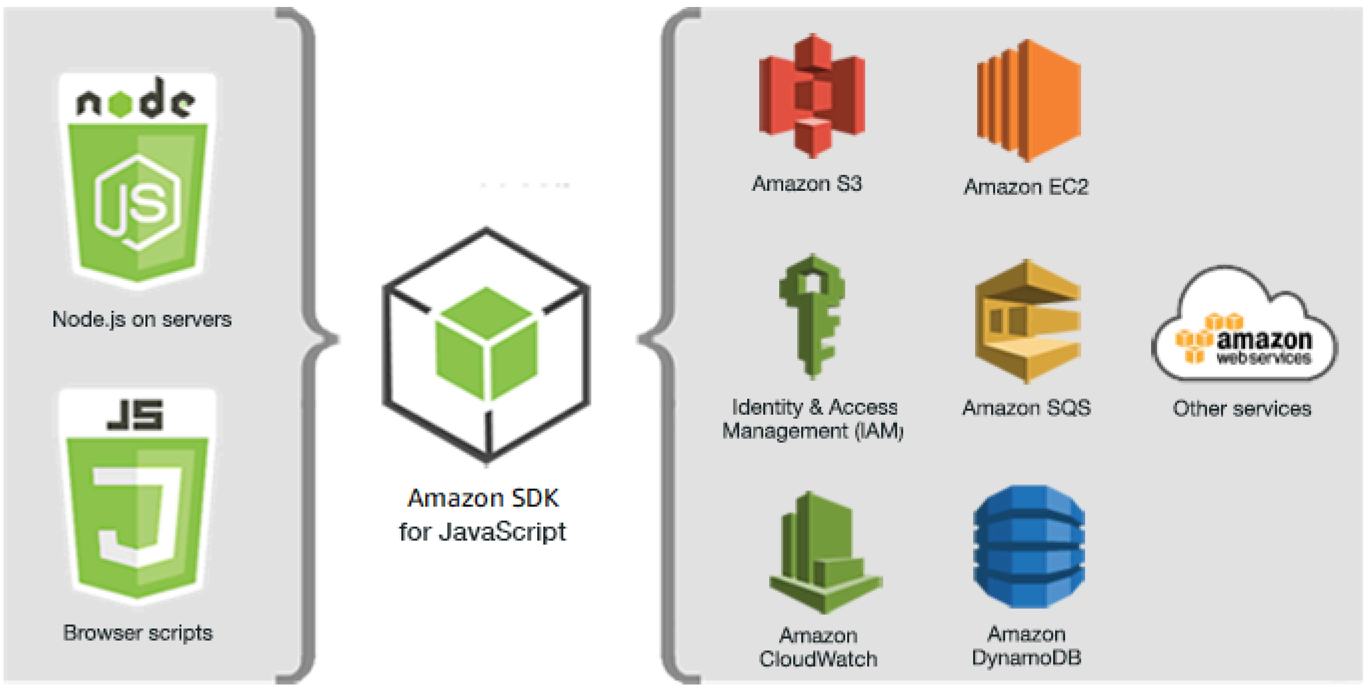
수신 규칙 사용 279
Amazon SNS 예제 285
주제 관리 285
주제에 메시지 게시 291
구독 관리 293
SMS 메시지 전송 299
Amazon SQS 예제 305
Amazon SQS에서 대기열 사용 306
Amazon SQS에서 메시지 전송 및 수신 310
Amazon SQS의 제한 시간 초과 관리 314
Amazon SQS에서 긴 폴링 활성화 316
Amazon SQS에서 DLQ(Dead Letter Queue) 사용 320
자습서 323
자습서: Amazon EC2 인스턴스에서 Node.js 설정 323
사전 조건 323
절차 323
Amazon Machine Image 생성 325
관련 리소스 325
API 참조 및 변경 사항 326
GitHub의 SDK 변경 로그 326
v3로 마이그레이션 327
보안 328
데이터 보호 328
자격 증명 및 액세스 관리 329
고객 330
보안 인증을 통한 인증 330
정책을 사용하여 액세스 관리 332
AWS 서비스에서 IAM을 사용하는 방식 333
AWS 보안 인증 및 액세스 문제 해결 333
규정 준수 검증 335
복원력 335
인프라 보안 336
TLS의 최소 버전 적용 337
Node.js에서 TLS 확인 및 적용 337
브라우저 스크립트에서 TLS 확인 및 적용 340
추가 리소스 342

AWS SDK 및 도구 참조 가이드 342
JavaScript SDK 포럼 342
GitHub의 JavaScript SDK 및 개발자 안내서 342
Gitter의 JavaScript SDK 342
문서 기록 343
문서 기록 343
이전 업데이트 344

AWS SDK for JavaScript v2가 지원 종료에 도달했습니다. AWS SDK for JavaScript v3로 마이그레이션하실 것을 권장합니다. 마이그레이션 방법에 대한 자세한 내용은 해당 공지 사항을 참조하세요.

AWS SDK for JavaScript란 무엇인가요?

AWS SDK for JavaScript에서는 AWS 서비스용 JavaScript API를 제공합니다. JavaScript API를 사용하여 Node.js용 또는 브라우저용 라이브러리 또는 애플리케이션을 빌드할 수 있습니다.



일부 서비스는 SDK에서 즉시 사용할 수 없습니다. AWS SDK for JavaScript에서 현재 지원되는 서비스를 알아보려면 https://github.com/aws/aws-sdk-js/blob/master/SERVICES.md를 참조하세요. GitHub의 SDK for JavaScript에 대한 자세한 내용은 추가 리소스를 참조하세요.

SDK 메이저 버전에 대한 유지 관리 및 지원

SDK 메이저 버전 및 기본 종속성의 유지 관리 및 지원에 대한 자세한 내용은 AWS SDK 및 도구 참조 안내서에서 다음 내용을 참조하세요.

- AWS SDK 및 도구 유지 관리 정책
- AWS SDK 및 도구 버전 지원 매트릭스

Node.js에서 SDK 사용

Node.js는 서버 측 JavaScript 애플리케이션을 실행하기 위한 교차 플랫폼 런타임입니다. 서버에서 실행할 Amazon EC2 인스턴스에서 Node.js를 설정할 수 있습니다. 또한 Node.js를 사용하여 온디맨드 AWS Lambda 함수를 작성할 수도 있습니다.

Node.js에 SDK를 사용하는 것은 웹 브라우저에서 JavaScript에 SDK를 사용하는 방법과 다릅니다. SDK를 로드하고 특정 웹 서비스에 액세스하는 데 필요한 자격 증명을 얻는 방법에서 차이가 비롯됩니다. 특정 API 사용이 Node.js와 브라우저 간에 다른 경우 해당 차이점이 표시됩니다.

AWS Amplify에서 SDK 사용

브라우저 기반 웹, 모바일 및 하이브리드 앱의 경우에도 GitHub의 AWS Amplify 라이브러리를 사용할 수 있습니다. 이 라이브러리는 SDK for JavaScript를 확장하여 선언형 인터페이스를 제공합니다.

Note

AWS Amplify와 같은 프레임워크는 SDK for JavaScript와 동일한 브라우저를 지원하지 않을 수 있습니다. 세부 정보는 프레임워크 설명서를 확인합니다.

웹 브라우저에서 SDK 사용

주요 웹 브라우저는 모두 JavaScript 확장을 지원합니다. 웹 브라우저에서 실행 중인 JavaScript 코드를 일반적으로 클라이언트 측 JavaScript라고 합니다.

웹 브라우저에서 SDK for JavaScript를 사용하는 방법은 Node.js에 SDK를 사용하는 방법과 다릅니다. SDK를 로드하고 특정 웹 서비스에 액세스하는 데 필요한 자격 증명을 얻는 방법에서 차이가 비롯됩니다. 특정 API 사용이 Node.js와 브라우저 간에 다른 경우 해당 차이점이 표시됩니다.

AWS SDK for JavaScript에서 지원되는 브라우저 목록은 지원되는 웹 브라우저 섹션을 참조하세요.

일반 사용 사례

브라우저 스크립트에서 SDK for JavaScript를 사용하면 여러 가지 흥미로운 사용 사례를 실현할 수 있습니다. 다음은 SDK for JavaScript를 사용하여 다양한 웹 서비스에 액세스함으로써 브라우저 애플리케이션에서 빌드할 수 있는 것에 대한 몇 가지 아이디어입니다.

- 조직 또는 프로젝트 요구 사항을 최대한 충족하기 위해 리전 및 서비스 전반에 걸쳐 기능에 액세스하고 기능을 결합하는 사용자 지정 콘솔을 AWS 서비스에 빌드합니다.
- Amazon Cognito 자격 증명을 사용하여 인증된 사용자가 Facebook 등의 타사 인증 사용을 포함해 브라우저 애플리케이션 및 웹 사이트에 액세스하도록 합니다.
- Amazon Kinesis를 사용하여 클릭 스트림 또는 기타 마케팅 데이터를 실시간으로 처리합니다.
- 웹 사이트 방문자 또는 애플리케이션 사용자에 대한 개별 사용자 기본 설정과 같은 서버리스 데이터 지속성에 Amazon DynamoDB를 사용합니다.
- AWS Lambda를 사용하여 지적 재산을 다운로드해 사용자에게 노출하는 일 없이 브라우저 스크립트에서 호출할 수 있는 독점 로직을 캡슐화합니다.

예제 정보

SDK for JavaScript 예제는 AWS 코드 샘플 라이브러리에서 찾아볼 수 있습니다.

AWS SDK for JavaScript 시작하기

AWS SDK for JavaScript에서는 브라우저 스크립트 또는 Node.js를 사용하여 웹 서비스에 액세스할 수 있습니다. 이 섹션에서는 각 JavaScript 환경에서 SDK for JavaScript를 사용하여 작업하는 방법을 보여주는 시작하기 연습 두 가지를 소개합니다.

주제

- 브라우저 스크립트에서 시작하기
- Node.js에서 시작하기

브라우저 스크립트에서 시작하기

이 브라우저 스크립트 예제는 다음을 보여 줍니다.

- Amazon Cognito 자격 증명을 사용하여 브라우저 스크립트에서 AWS 서비스에 액세스하는 방법
- Amazon Polly를 사용하여 텍스트를 합성된 음성으로 변환하는 방법
- presigner 객체를 사용하여 미리 서명된 URL을 생성하는 방법

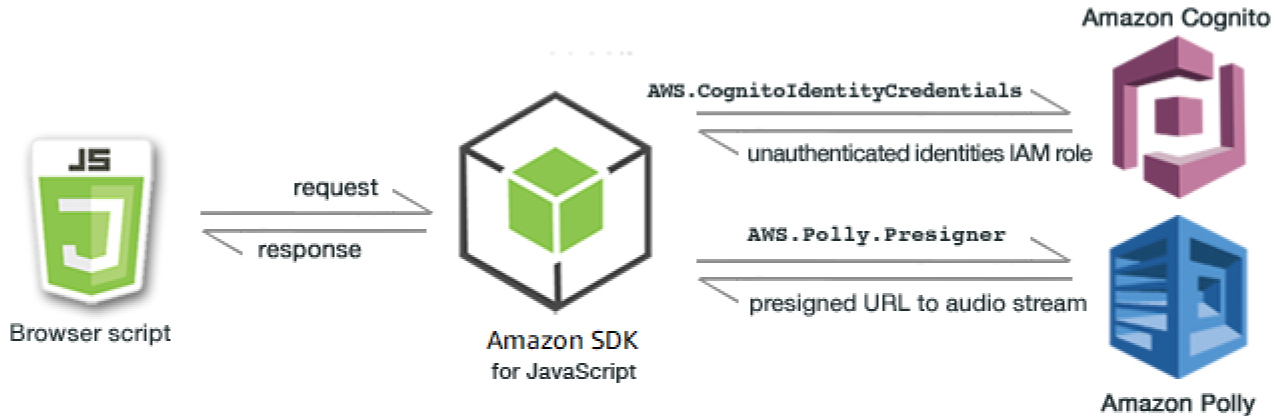
시나리오

Amazon Polly는 텍스트를 생생한 스피치로 변환하는 클라우드 서비스입니다. Amazon Polly를 사용하여 참여도와 접근성을 높이는 애플리케이션을 개발할 수 있습니다. Amazon Polly는 여러 언어를 지원하며 다양하고 생생한 음성을 포함합니다. Amazon Polly에 대한 자세한 내용은 Amazon Polly 개발자 안내서를 참조하세요.

이 예제에서는 입력한 텍스트를 가져와 Amazon Polly로 보낸 다음, 재생할 텍스트의 합성된 오디오의 URL을 반환하는 간단한 브라우저 스크립트를 설정하고 실행하는 방법을 보여줍니다. 브라우저 스크립트는 Amazon Cognito 자격 증명을 사용하여 AWS 서비스에 액세스하는 데 필요한 자격 증명을 제공합니다. 브라우저 스크립트에서 SDK for JavaScript를 로드해 사용하는 기본 패턴이 보입니다.

Note

이 예제에서 합성된 스피치를 재생하려면 HTML 5 오디오를 지원하는 브라우저에서 실행해야 합니다.



브라우저 스크립트는 SDK for JavaScript를 사용하여 다음 API를 통해 텍스트를 합성합니다.

- `AWS.CognitoIdentityCredentials` 생성자
- `AWS.Polly.Presigner` 생성자
- `getSynthesizeSpeechUrl`

1단계: Amazon Cognito 자격 증명 풀 생성

이 예제에서는 Amazon Cognito 자격 증명 풀을 생성 후 이를 사용해 Amazon Polly 서비스에 대한 미인증 액세스 권한을 브라우저 스크립트에 제공합니다. 또한 자격 증명 풀을 생성하면 IAM 역할이 2개 생성되는데, 하나는 자격 증명 공급자가 인증한 사용자를 지원하고, 다른 하나는 인증되지 않은 게스트 사용자를 지원합니다.

이 연습에서는 인증되지 않은 사용자 역할만 사용해 작업하여 작업에 집중할 수 있도록 합니다. 자격 증명 공급자 및 인증된 사용자에 대한 지원은 나중에 통합할 수 있습니다. Amazon Cognito 자격 증명 풀 추가에 관한 자세한 내용은 Amazon Cognito 개발자 안내서의 자습서: 자격 증명 풀 생성 섹션을 참조하세요.

Amazon Cognito 자격 증명 풀 생성

1. AWS Management Console에 로그인한 후 https://console.aws.amazon.com/cognito/에서 Amazon Cognito 콘솔을 엽니다.
2. 왼쪽 탐색 창에서 자격 증명 풀을 선택합니다.
3. 자격 증명 풀 생성을 선택합니다.
4. 자격 증명 풀 신뢰 구성에서 사용자 인증을 위한 게스트 액세스를 선택합니다.
5. 권한 구성에서 새 IAM 역할 생성을 선택하고 IAM 역할 이름에 이름(예: getStartedRole)을 입력합니다.
6. 속성 구성에서 자격 증명 풀 이름에 이름(예: getStartedPool)을 입력합니다.
7. 검토 및 생성에서 새 자격 증명 풀에 대한 선택 사항을 확인합니다. 편집을 선택하여 마법사로 돌아가서 설정을 변경합니다. 완료하면 자격 증명 풀 생성을 선택합니다.
8. 새로 생성된 Amazon Cognito 자격 증명 풀의 자격 증명 풀 ID 및 리전을 기록해 둡니다. 이러한 값은 4단계: 브라우저 스크립트 작성에서 *IDENTITY_POOL_ID* 및 *REGION*을 바꾸는 데 필요합니다.

Amazon Cognito 자격 증명 풀을 생성하면 브라우저 스크립트에 필요한 Amazon Polly에 대한 권한을 추가할 준비가 된 것입니다.

2단계: 생성된 IAM 역할에 정책 추가

음성 합성을 위해 브라우저 스크립트가 Amazon Polly에 액세스하도록 하려면 Amazon Cognito 자격 증명 풀에 대해 생성된 인증되지 않은 IAM 역할을 사용합니다. 이 단계를 수행하려면 해당 역할에 IAM 정책을 추가해야 합니다. IAM 역할 수정에 관한 자세한 내용은 IAM 사용 설명서의 역할 권한 정책 수정 섹션을 참조하세요.

미인증 사용자와 연결된 IAM 역할에 Amazon Polly 정책을 추가하는 방법

1. AWS Management Console에 로그인하여 https://console.aws.amazon.com/iam/에서 IAM 콘솔을 엽니다.
2. 왼쪽 탐색 창에서 역할을 선택합니다.
3. 수정하려는 역할의 이름(예: getStartedRole)을 선택한 다음, 권한 탭을 선택합니다.
4. 권한 추가를 선택한 다음, 정책 연결을 선택합니다.
5. 이 역할의 권한 추가 페이지에서 AmazonPollyReadOnly 확인란을 찾아 선택합니다.

Note

이 프로세스를 사용하여 AWS 서비스에 대한 액세스를 활성화할 수 있습니다.

6. 권한 추가를 선택합니다.

Amazon Cognito 자격 증명 풀을 생성하고 인증되지 않은 사용자에 대한 IAM 역할에 Amazon Polly에 대한 권한을 추가하면 웹페이지 및 브라우저 스크립트를 작성할 준비가 된 것입니다.

3단계: HTML 페이지 만들기

이 샘플 앱은 사용자 인터페이스와 브라우저 스크립트가 포함된 단일 HTML 페이지로 구성되어 있습니다. 시작하려면 HTML 문서를 생성하고 다음 내용을 복사해 생성한 HTML 문서에 복사합니다. 이 페이지에는 입력 필드 및 버튼, 합성된 스피치 재생을 위한 <audio> 요소와 메시지 표시를 위한 <p> 요소가 포함되어 있습니다. 전체 예제는 이 페이지 맨 아래에 나와 있습니다.

<audio> 요소에 대한 자세한 내용은 오디오를 참조하세요.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>AWS SDK for JavaScript - Browser Getting Started Application</title>
  </head>

  <body>
    <div id="textToSynth">
      <input autofocus size="23" type="text" id="textEntry" value="It's very good to
 meet you."/>
      <button class="btn default" onClick="speakText()">Synthesize</button>
      <p id="result">Enter text above then click Synthesize</p>
    </div>
    <audio id="audioPlayback" controls>
      <source id="audioSource" type="audio/mp3" src="">
    </audio>
    <!-- (script elements go here) -->
 </body>
</html>
```

HTML 파일의 이름을 polly.html로 지정하여 저장합니다. 애플리케이션에 대한 사용자 인터페이스를 생성하면 이 애플리케이션을 실행할 브라우저 스크립트 코드를 추가할 준비가 된 것입니다.

4단계: 브라우저 스크립트 작성

브라우저 스크립트를 작성할 때 가장 먼저 할 일은 페이지에서 `<script>` 요소 뒤에 `<audio>` 요소를 추가하여 SDK for JavaScript를 포함하는 것입니다. AWS SDK for JavaScript API 참조 가이드의 SDK for JavaScript에 대한 API 참조 섹션에서 현재 SDK_VERSION_NUMBER를 찾을 수 있습니다.

```
<script src="https://sdk.amazonaws.com/js/aws-sdk-SDK_VERSION_NUMBER.min.js"></script>
```

그런 다음 SDK 항목 뒤에 새 `<script type="text/javascript">` 요소를 추가합니다. 이 요소에 브라우저 스크립트를 추가합니다. SDK에 대한 AWS 리전 및 자격 증명을 설정합니다. 다음으로, 버튼이 이벤트 핸들러로 호출할 `speakText()` 함수를 생성합니다.

Amazon Polly를 사용해 음성을 합성하려면 출력 사운드 형식, 샘플링 비율, 사용할 음성의 ID 및 재생할 텍스트 등의 파라미터를 제공해야 합니다 처음에 이러한 파라미터를 생성할 때에는 `Text:` 파라미터는 빈 문자열로 설정합니다. `Text:` 파라미터는 웹페이지의 `<input>` 요소에서 가져오는 값으로 설정됩니다. 다음 코드의 *IDENTITY_POOL_ID*와 *##*을 1단계: Amazon Cognito 자격 증명 풀 생성에 기록된 값으로 바꿉니다.

```
    <script type="text/javascript">

        // Initialize the Amazon Cognito credentials provider
        AWS.config.region = 'REGION';
        AWS.config.credentials = new AWS.CognitoIdentityCredentials({IdentityPoolId:
 'IDENTITY_POOL_ID'});

        // Function invoked by button click
        function speakText() {
            // Create the JSON parameters for getSynthesizeSpeechUrl
            var speechParams = {
                OutputFormat: "mp3",
                SampleRate: "16000",
                Text: "",
                TextType: "text",
                VoiceId: "Matthew"
            };
            speechParams.Text = document.getElementById("textEntry").value;
```

Amazon Polly가 합성된 음성을 오디오 스트림으로 반환합니다. 이 오디오를 브라우저에서 재생하는 가장 쉬운 방법은 Amazon Polly가 미리 서명된 URL에서 해당 오디오를 사용할 수 있도록 만드는 것입니다. 그러면 웹페이지에서 `<audio>` 요소의 `src` 속성을 설정할 수 있습니다.

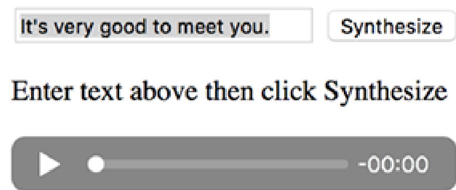
새 `AWS.Polly` 서비스 객체를 생성합니다. 그런 다음 합성된 스피치 오디오를 검색할 수 있는 미리 서명된 URL을 생성하는 데 사용할 `AWS.Polly.Presigner` 객체를 생성합니다. 정의한 스피치 파라미터와 `AWS.Polly` 생성자에 대해 생성한 `AWS.Polly.Presigner` 서비스 객체를 전달해야 합니다.

presigner 객체를 생성한 후에는 해당 객체의 `getSynthesizeSpeechUrl` 메서드를 호출해 스피치 파라미터를 전달합니다. 성공하면 이 메서드는 합성된 스피치의 URL을 반환합니다. 그런 다음 재생을 위해 이 URL을 `<audio>` 요소에 할당합니다.

```
            // Create the Polly service object and presigner object
            var polly = new AWS.Polly({apiVersion: '2016-06-10'});
            var signer = new AWS.Polly.Presigner(speechParams, polly)

            // Create presigned URL of synthesized speech file
            signer.getSynthesizeSpeechUrl(speechParams, function(error, url) {
            if (error) {
                document.getElementById('result').innerHTML = error;
            } else {
                document.getElementById('audioSource').src = url;
                document.getElementById('audioPlayback').load();
                document.getElementById('result').innerHTML = "Speech ready to play.";
            }
          });
        }
    </script>
```

5단계: 샘플 실행

샘플 앱을 실행하려면 웹 브라우저로 `polly.html`을 로드합니다. 그러면 브라우저의 모양이 다음과 같아야 합니다.



입력 상자에 스피치로 변환하려는 문구를 입력한 다음 Synthesize(합성)를 선택합니다. 오디오가 재생 준비가 되면 메시지가 나타납니다. 오디오 플레이어 컨트롤을 사용하여 합성된 스피치를 들을 수 있습니다.

전체 샘플

다음은 브라우저 스크립트가 포함된 전체 HTML 페이지입니다. 이 페이지는 GitHub에서도 사용할 수 있습니다.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>AWS SDK for JavaScript - Browser Getting Started Application</title>
  </head>

  <body>
    <div id="textToSynth">
      <input autofocus size="23" type="text" id="textEntry" value="It's very good to
 meet you."/>
      <button class="btn default" onClick="speakText()">Synthesize</button>
      <p id="result">Enter text above then click Synthesize</p>
    </div>
    <audio id="audioPlayback" controls>
      <source id="audioSource" type="audio/mp3" src="">
    </audio>
    <script src="https://sdk.amazonaws.com/js/aws-sdk-2.410.0.min.js"></script>
    <script type="text/javascript">

        // Initialize the Amazon Cognito credentials provider
        AWS.config.region = 'REGION';
        AWS.config.credentials = new AWS.CognitoIdentityCredentials({IdentityPoolId:
 'IDENTITY_POOL_ID'});

        // Function invoked by button click
        function speakText() {
            // Create the JSON parameters for getSynthesizeSpeechUrl
            var speechParams = {
                OutputFormat: "mp3",
                SampleRate: "16000",
                Text: "",
                TextType: "text",
                VoiceId: "Matthew"
```

```
            };
            speechParams.Text = document.getElementById("textEntry").value;

            // Create the Polly service object and presigner object
            var polly = new AWS.Polly({apiVersion: '2016-06-10'});
            var signer = new AWS.Polly.Presigner(speechParams, polly)

            // Create presigned URL of synthesized speech file
            signer.getSynthesizeSpeechUrl(speechParams, function(error, url) {
            if (error) {
                document.getElementById('result').innerHTML = error;
            } else {
                document.getElementById('audioSource').src = url;
                document.getElementById('audioPlayback').load();
                document.getElementById('result').innerHTML = "Speech ready to play.";
            }
          });
        }
    </script>
  </body>
</html>
```

가능한 개선 사항

브라우저 스크립트에서의 SDK for JavaScript 활용도를 높일 수 있는 애플리케이션을 변형할 수 있습니다.

- 다른 사운드 출력 형식으로 실험합니다.
- Amazon Polly에서 제공하는 다양한 음성을 선택할 수 있는 옵션을 추가합니다.
- 인증된 IAM 역할과 함께 사용할 수 있도록 Facebook 또는 Amazon 등과 같은 자격 증명 공급자를 통합합니다.

Node.js에서 시작하기

이 Node.js 코드 예제는 다음을 보여 줍니다.

- 프로젝트를 위한 `package.json` 매니페스트를 생성하는 방법
- 프로젝트에서 사용하는 모듈을 설치 및 포함하는 방법
- `AWS.S3` 클라이언트 클래스에서 Amazon Simple Storage Service(S3) 서비스 객체를 생성하는 방법
- Amazon S3 버킷을 생성하고 해당 버킷에 객체를 업로드하는 방법

시나리오

이 예제에서는 Amazon S3 버킷을 생성한 다음 이 버킷에 텍스트 객체를 추가하는 Node.js 모듈을 간단하게 설정 및 실행하는 방법을 보여줍니다.

Amazon S3의 버킷 이름은 전역적으로 고유해야 하기 때문에 이 예제에는 버킷 이름에 통합할 수 있는 고유한 ID 값을 생성하는 타사 Node.js 모듈이 포함되어 있습니다. 이 추가 모듈의 이름은 `uuid`입니다.

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 이러한 작업들을 완료해야 합니다.

- Node.js 모듈 개발을 위한 작업 디렉터리를 생성합니다. 이 디렉터리의 이름을 `awsnodesample`로 지정합니다. 디렉터리는 애플리케이션에서 업데이트할 수 있는 위치에 생성해야 합니다. 예를 들어 Windows에서는 "C:\Program Files" 아래에 디렉터리를 생성하지 마십시오.
- Node.js를 설치합니다. 자세한 내용은 [Node.js](https://nodejs.org) 웹 사이트를 참조하세요. https://nodejs.org/en/download/current/에서는 다양한 운영 체제에 맞는 Node.js의 최신 버전 및 LTS 버전 다운로드를 찾을 수 있습니다.

목차

- 1단계: SDK 및 종속성 설치
- 2단계: 자격 증명 구성
- 3단계: 프로젝트용 패키지 JSON 생성
- 4단계: Node.js 작성
- 5단계: 샘플 실행

1단계: SDK 및 종속성 설치

Node.js 패키지 관리자인 npm으로 SDK for JavaScript를 설치합니다.

이 패키지의 `awsnodesample` 디렉터리에서 명령줄에 다음을 입력합니다.

```
npm install aws-sdk
```

이 명령은 프로젝트에 SDK for JavaScript를 설치하고 package.json을 업데이트하여 SDK를 프로젝트 종속성으로 나열합니다. npm 웹 사이트에서 "aws-sdk"를 검색하여 이 패키지에 대한 정보를 찾을 수 있습니다.

다음으로, 명령줄에 다음을 입력하여 프로젝트에 uuid 모듈을 설치합니다. 그러면 모듈이 설치되고 package.json이 업데이트됩니다. uuid에 대한 자세한 내용은 https://www.npmjs.com/package/uuid에서 모듈 페이지를 참조하세요.

```
npm install uuid
```

이러한 패키지와 연결된 코드는 프로젝트의 node_modules 하위 디렉터리에 설치됩니다.

Node.js 패키지 설치에 대한 자세한 내용은 로컬에서 패키지 다운로드 및 설치 및 npm(Node.js 패키지 관리자) 웹 사이트의 Node.js 모듈 생성 섹션을 참조하세요. AWS SDK for JavaScript 설치 및 다운로드에 대한 자세한 내용은 SDK for JavaScript 설치 섹션을 참조하세요.

2단계: 자격 증명 구성

SDK에서 계정과 해당 계정의 리소스에만 액세스할 수 있도록 AWS에 자격 증명을 제공해야 합니다. 계정 자격 증명을 받는 방법에 대한 자세한 내용은 AWS를 사용한 SDK 인증 섹션을 참조하세요.

이러한 정보를 보관하려면 공유 자격 증명 파일을 만드는 것이 좋습니다. 자세한 방법은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드을 참조하세요. 자격 증명 파일의 모양은 다음 예와 유사해야 합니다.

```
[default]
aws_access_key_id = YOUR_ACCESS_KEY_ID
aws_secret_access_key = YOUR_SECRET_ACCESS_KEY
```

Node.js로 다음 코드를 실행하여 자격 증명을 올바르게 설정했는지 여부를 확인할 수 있습니다.

```
var AWS = require("aws-sdk");

AWS.config.getCredentials(function(err) {
  if (err) console.log(err.stack);
  // credentials not loaded
  else {
    console.log("Access key:", AWS.config.credentials.accessKeyId);
```

```
  }
});
```

마찬가지로, config 파일에서 리전을 올바르게 설정한 경우 AWS_SDK_LOAD_CONFIG 환경 변수를 임의의 값으로 설정하고 다음 코드를 사용하여 해당 값을 표시할 수 있습니다.

```
var AWS = require("aws-sdk");

console.log("Region: ", AWS.config.region);
```

3단계: 프로젝트용 패키지 JSON 생성

awsnodesample 프로젝트 디렉터리를 생성한 다음 Node.js 프로젝트에 대한 메타데이터를 보관하기 위한 package.json 파일을 만들어 추가합니다. Node.js 프로젝트에서 package.json을 사용하는 방법에 관한 자세한 내용은 패키지 .json 파일 생성을 참조하세요.

프로젝트 디렉터리에서 package.json이라는 새 파일을 생성합니다. 그런 다음 이 파일에 다음 JSON을 추가합니다.

```
{
  "dependencies": {},
  "name": "aws-nodejs-sample",
  "description": "A simple Node.js application illustrating usage of the SDK for
 JavaScript.",
  "version": "1.0.1",
  "main": "sample.js",
  "devDependencies": {},
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "NAME",
  "license": "ISC"
}
```

파일을 저장합니다. 필요한 모듈을 설치하면 이 파일의 dependencies 부분이 완료됩니다. GitHub에서 이러한 종속성의 예제를 보여주는 JSON 파일을 찾을 수 있습니다.

4단계: Node.js 작성

예제 코드를 포함할 sample.js라는 새 파일을 만듭니다. require 함수 호출을 추가하고 사용할 수 있도록 SDK for JavaScript 및 uuid 모듈을 포함하는 것으로 시작합니다.

인식 가능한 접두사(이 경우에는 'node-sdk-sample-')에 고유한 ID 값을 추가하여 Amazon S3 버킷을 생성하는 데 사용할 고유한 버킷 이름을 작성합니다. uuid 모듈을 호출하여 고유한 ID를 생성합니다. 그런 다음 버킷에 객체를 업로드하는 데 사용하는 Key 파라미터의 이름을 생성합니다.

promise 객체를 생성하여 createBucket 서비스 객체의 AWS.S3 메서드를 호출합니다. 성공적인 응답 시 새로 생성한 버킷에 텍스트를 업로드하는 데 필요한 파라미터를 생성합니다. 다른 promise를 사용하여 putObject 메서드를 호출해 버킷에 텍스트 객체를 업로드합니다.

```
// Load the SDK and UUID
var AWS = require("aws-sdk");
var uuid = require("uuid");

// Create unique bucket name
var bucketName = "node-sdk-sample-" + uuid.v4();
// Create name for uploaded object key
var keyName = "hello_world.txt";

// Create a promise on S3 service object
var bucketPromise = new AWS.S3({ apiVersion: "2006-03-01" })
  .createBucket({ Bucket: bucketName })
  .promise();

// Handle promise fulfilled/rejected states
bucketPromise
  .then(function (data) {
    // Create params for putObject call
    var objectParams = {
      Bucket: bucketName,
      Key: keyName,
      Body: "Hello World!",
    };
    // Create object upload promise
    var uploadPromise = new AWS.S3({ apiVersion: "2006-03-01" })
      .putObject(objectParams)
      .promise();
    uploadPromise.then(function (data) {
      console.log(
        "Successfully uploaded data to " + bucketName + "/" + keyName
      );
    });
  })
  .catch(function (err) {
    console.error(err, err.stack);
```

```
});
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

5단계: 샘플 실행

다음 명령을 입력하여 샘플을 실행합니다.

```
node sample.js
```

업로드에 성공하면 명령줄에 확인 메시지가 표시됩니다. 또한 Amazon S3 콘솔에서도 버킷 및 업로드된 텍스트 객체를 찾을 수 있습니다.

SDK for JavaScript 설정

이 섹션에서는 웹 브라우저 및 Node.js에서 사용할 SDK for JavaScript를 설치하는 방법을 설명합니다. 또한 SDK에서 지원되는 웹 서비스에 액세스할 수 있도록 SDK를 로드하는 방법도 보여 줍니다.

Note

React Native 개발자는 AWS Amplify를 사용하여 AWS에 새로운 프로젝트를 생성해야 합니다. 자세한 내용은 aws-sdk-react-native 아카이브를 참조하세요.

주제

- 사전 조건
- SDK for JavaScript 설치
- SDK for JavaScript 로드
- SDK for JavaScript 버전 1에서 업그레이드

사전 조건

AWS SDK for JavaScript를 사용하기 전에 코드를 Node.js에서 실행해야 하는지 또는 웹 브라우저에서 실행해야 하는지를 결정합니다. 그런 후 다음을 수행합니다.

- Node.js의 경우 아직 설치되어 있지 않으면 서버에 Node.js를 설치합니다.
- 웹 브라우저의 경우 지원해야 하는 브라우저 버전을 식별합니다.

주제

- AWS Node.js 환경 설정
- 지원되는 웹 브라우저

AWS Node.js 환경 설정

애플리케이션을 실행할 수 있는 AWS Node.js 환경을 설정하려면 다음 방법 중 하나를 사용합니다.

- Node.js가 미리 설치되어 있는 Amazon Machine Image(AMI)를 선택하고 해당 AMI를 사용하여 Amazon EC2 인스턴스를 생성합니다. Amazon EC2 인스턴스를 생성할 때 AWS Marketplace에서

AMI를 선택합니다. AWS Marketplace에서 Node.js를 검색하고 Node.js의 한 버전(32비트 또는 64비트)이 미리 설치되어 있는 AMI 옵션을 선택합니다.

- Amazon EC2 인스턴스를 생성하고 해당 인스턴스에 Node.js를 설치합니다. Amazon Linux 인스턴스에 Node.js를 설치하는 방법에 관한 자세한 내용은 자습서: Amazon EC2 인스턴스에서 Node.js 설정 단원을 참조하세요.
- AWS Lambda를 사용하여 서버리스 환경을 생성하고 Node.js를 Lambda 함수로 실행합니다. Lambda 함수 내에서 Node.js를 사용하는 방법에 관한 자세한 내용은 AWS Lambda 개발자 안내서의 프로그래밍 모델(Node.js) 섹션을 참조하세요.
- Node.js 애플리케이션을 AWS Elastic Beanstalk에 배포합니다. Elastic Beanstalk로 Node.js를 사용하는 방법에 관한 자세한 내용은 AWS Elastic Beanstalk 개발자 안내서의 AWS Elastic Beanstalk에 Node.js 애플리케이션 배포 섹션을 참조하세요.

지원되는 웹 브라우저

SDK for JavaScript는 다음 최소 버전을 포함하여 모든 최신 웹 브라우저를 지원합니다.

브라우저	버전
Google Chrome	28.0 이상
Mozilla Firefox	26.0 이상
Opera	17.0 이상
Microsoft Edge	25.10 이상
Windows Internet Explorer	해당 사항 없음
Apple Safari	5 이상
Android 브라우저	4.3 이상

> **Note**
>
> AWS Amplify와 같은 프레임워크는 SDK for JavaScript와 동일한 브라우저를 지원하지 않을 수 있습니다. 세부 정보는 프레임워크 설명서를 확인합니다.

SDK for JavaScript 설치

AWS SDK for JavaScript 설치 여부와 방법은 코드가 Node.js 모듈에서 실행하는지 또는 브라우저 스크립트에서 실행하는지에 따라 다릅니다.

일부 서비스는 SDK에서 즉시 사용할 수 없습니다. AWS SDK for JavaScript에서 현재 지원되는 서비스를 알아보려면 https://github.com/aws/aws-sdk-js/blob/master/SERVICES.md를 참조하세요.

Node

Node.js용 AWS SDK for JavaScript를 설치하는 기본 방법은 npm(Node.js 패키지 관리자)을 사용하는 것입니다. 이렇게 하려면 명령줄에 다음을 입력합니다.

```
npm install aws-sdk
```

다음 오류 메시지가 나타나는 경우

```
npm WARN deprecated node-uuid@1.4.8: Use uuid module instead
```

명령줄에 다음 명령을 입력합니다.

```
npm uninstall --save node-uuid
npm install --save uuid
```

Browser

브라우저 스크립트에서 사용하려면 SDK를 설치할 필요가 없습니다. HTML 페이지의 스크립트를 사용하여 Amazon Web Services에서 호스팅된 SDK 패키지를 직접 로드할 수 있습니다. 호스팅된 SDK 패키지는 CORS(교차 오리진 리소스 공유)를 적용하는 AWS 서비스 중 일부를 지원합니다. 자세한 내용은 SDK for JavaScript 로드 섹션을 참조하세요.

사용할 특정 웹 서비스와 버전을 선택하는 SDK의 사용자 지정 빌드를 생성할 수 있습니다. 그런 다음 로컬 개발을 위해 사용자 지정 SDK 패키지를 다운로드하고 애플리케이션에서 사용할 수 있도록 호스팅합니다. SDK의 사용자 지정 빌드 생성에 대한 자세한 내용은 브라우저용 SDK 빌드 섹션을 참조하세요.

GitHub의 다음 위치에서 최신 AWS SDK for JavaScript의 축소형 및 비축소형 배포 가능 버전을 다운로드할 수 있습니다.

https://github.com/aws/aws-sdk-js/tree/master/dist

Bower를 사용한 설치

Bower는 웹용 패키지 관리자입니다. Bower를 설치한 후에는 Bower를 사용하여 SDK를 설치할 수 있습니다. Bower를 사용하여 SDK를 설치하려면 터미널 창에 다음을 입력합니다.

```
bower install aws-sdk-js
```

SDK for JavaScript 로드

SDK for JavaScript를 로드하는 방법은 웹 브라우저에서 실행하기 위해 로드하는지 또는 Node.js에서 실행하기 위해 로드하는지에 따라 다릅니다.

일부 서비스는 SDK에서 즉시 사용할 수 없습니다. AWS SDK for JavaScript에서 현재 지원되는 서비스를 알아보려면 https://github.com/aws/aws-sdk-js/blob/master/SERVICES.md를 참조하세요.

Node.js

SDK를 설치한 후 `require`를 사용하여 노드 애플리케이션에서 AWS 패키지를 로드할 수 있습니다.

```
var AWS = require('aws-sdk');
```

React Native

React Native 프로젝트에서 SDK를 사용하려면 먼저 npm을 사용하여 SDK를 설치합니다.

```
npm install aws-sdk
```

애플리케이션에서 다음 코드를 사용하여 SDK의 React Native 호환 버전을 참조합니다.

```
var AWS = require('aws-sdk/dist/aws-sdk-react-native');
```

Browser

SDK를 시작하는 가장 빠른 방법은 호스팅된 SDK 패키지를 Amazon Web Services에서 직접 로드하는 것입니다. 이를 위해 다음 형식으로 `<script>` 요소를 HTML 페이지에 추가합니다.

```
<script src="https://sdk.amazonaws.com/js/aws-sdk-SDK_VERSION_NUMBER.min.js"></
script>
```

AWS SDK for JavaScript API 참조 가이드의 SDK for JavaScript에 대한 API 참조 섹션에서 현재 SDK_VERSION_NUMBER를 찾을 수 있습니다.

SDK가 페이지에 로드된 후에는 글로벌 변수 `AWS`(또는 `window.AWS`)에서 SDK를 사용할 수 있습니다.

browserify를 사용하여 코드 및 모듈 종속성을 번들링하는 경우 Node.js에서 수행하는 것과 같은 방식으로 `require`를 사용하여 SDK를 로드합니다.

SDK for JavaScript 버전 1에서 업그레이드

다음 메모를 참조해 SDK for JavaScript를 버전 1에서 버전 2로 업그레이드하세요.

입력/출력 시 Base64 및 타임스탬프의 자동 변환

이제 SDK는 사용자를 대신하여 base64 인코딩 값과 타임스탬프 값을 자동으로 인코딩합니다. 이 변경 사항은 base64 또는 타임스탬프 값이 요청에 따라 전송되거나 base64 인코딩 값을 고려하는 응답에서 반환되는 작업에 영향을 미칩니다.

이전에 base64로 변환된 사용자 코드는 더 이상 필요하지 않습니다. base64로 인코딩된 값은 이제 서버 응답에서 버퍼 객체로 반환되며 버퍼 입력으로 전달될 수도 있습니다. 예를 들어, 다음 버전 1 `SQS.sendMessage` 파라미터는 다음을 수행할 수 있습니다.

```
var params = {
   MessageBody: 'Some Message',
   MessageAttributes: {
      attrName: {
         DataType: 'Binary',
         BinaryValue: new Buffer('example text').toString('base64')
      }
   }
};
```

다음과 같이 다시 작성할 수 있습니다.

```
var params = {
   MessageBody: 'Some Message',
   MessageAttributes: {
      attrName: {
         DataType: 'Binary',
         BinaryValue: 'example text'
```

```
      }
    }
};
```

메시지를 읽는 방법은 다음과 같습니다.

```
sqs.receiveMessage(params, function(err, data) {
  // buf is <Buffer 65 78 61 6d 70 6c 65 20 74 65 78 74>
  var buf = data.Messages[0].MessageAttributes.attrName.BinaryValue;
  console.log(buf.toString()); // "example text"
});
```

response.data.RequestId를 response.requestId로 이동

이제 SDK는 `response` 속성 내부보다는 `response.data` 객체의 일관적인 장소에 모든 서버의 요청 ID를 저장합니다. 이렇게 하면 다양한 방식으로 요청 ID를 공개하는 서비스 전체에서 일관성을 향상할 수 있습니다. 또한 이 변경은 `response.data.RequestId` 속성의 이름을 `response.requestId`(콜백 함수 내부에서는 `this.requestId`)로 바꾸는 획기적인 변경입니다.

코드에서 다음을 변경합니다.

```
svc.operation(params, function (err, data) {
  console.log('Request ID:', data.RequestId);
});
```

다음의 것들로 변경됩니다.

```
svc.operation(params, function () {
  console.log('Request ID:', this.requestId);
});
```

노출된 래퍼 요소

`AWS.ElastiCache`, `AWS.RDS` 또는 `AWS.Redshift`를 사용하는 경우 일부 작업에 대한 응답으로 최상위 수준 출력 속성을 통해 응답에 액세스해야 합니다.

예를 들어, `RDS.describeEngineDefaultParameters` 메서드는 다음을 반환하는 데 사용됩니다.

```
{ Parameters: [ ... ] }
```

이제 다음을 반환합니다.

```
{ EngineDefaults: { Parameters: [ ... ] } }
```

각 서비스별로 영향을 받는 작업의 목록은 다음 표와 같습니다.

클라이언트 클래스	작업
AWS.ElastiCache	authorizeCacheSecurityGroupIngress createCacheCluster createCacheParameterGroup createCacheSecurityGroup createCacheSubnetGroup createReplicationGroup deleteCacheCluster deleteReplicationGroup describeEngineDefaultParameters modifyCacheCluster modifyCacheSubnetGroup modifyReplicationGroup purchaseReservedCacheNodesOffering rebootCacheCluster revokeCacheSecurityGroupIngress
AWS.RDS	addSourceIdentifierToSubscription

클라이언트 클래스	작업
	authorizeDBSecurityGroupIngress copyDBSnapshot createDBInstance createDBInstanceReadReplica createDBParameterGroup createDBSecurityGroup createDBSnapshot createDBSubnetGroup createEventSubscription createOptionGroup deleteDBInstance deleteDBSnapshot deleteEventSubscription describeEngineDefaultParameters modifyDBInstance modifyDBSubnetGroup modifyEventSubscription modifyOptionGroup promoteReadReplica purchaseReservedDBInstances Offering rebootDBInstance

클라이언트 클래스	작업
	removeSourceIdentifierFromSubscription restoreDBInstanceFromDBSnapshot restoreDBInstanceToPointInTime revokeDBSecurityGroupIngress

클라이언트 클래스	작업
AWS.Redshift	authorizeClusterSecurityGroupIngress authorizeSnapshotAccess copyClusterSnapshot createCluster createClusterParameterGroup createClusterSecurityGroup createClusterSnapshot createClusterSubnetGroup createEventSubscription createHsmClientCertificate createHsmConfiguration deleteCluster deleteClusterSnapshot describeDefaultClusterParameters disableSnapshotCopy enableSnapshotCopy modifyCluster modifyClusterSubnetGroup modifyEventSubscription modifySnapshotCopyRetentionPeriod

클라이언트 클래스	작업
	purchaseReservedNodeOffering rebootCluster restoreFromClusterSnapshot revokeClusterSecurityGroupIngress revokeSnapshotAccess rotateEncryptionKey

삭제된 클라이언트 속성

`.Client` 및 `.client` 속성은 서비스 객체에서 제거되었습니다. 서비스 클래스에서 `.Client` 속성을 사용하거나 서비스 객체 인스턴스에서 `.client` 속성을 사용하는 경우 이러한 속성을 코드에서 제거하십시오.

SDK for JavaScript 버전 1에서 사용되는 코드:

```
var sts = new AWS.STS.Client();
// or
var sts = new AWS.STS();

sts.client.operation(...);
```

다음 코드로 변경해야 합니다.

```
var sts = new AWS.STS();
sts.operation(...)
```

SDK for JavaScript 구성

API를 사용하여 웹 서비스를 간접 호출하려면 SDK for JavaScript를 사용하기 전에 SDK를 구성해야 합니다. 최소한 다음 설정을 구성해야 합니다.

- 서비스를 요청할 리전
- SDK 리소스에 대한 액세스 권한을 부여하는 인증 자격 증명

이러한 설정 외에도 AWS 리소스에 대한 권한도 구성해야 할 수 있습니다. 예를 들어 Amazon S3 버킷에 대한 액세스를 제한하거나 읽기 전용 액세스만 가능하도록 Amazon DynamoDB 테이블을 제한할 수 있습니다.

AWS SDK 및 도구 참조 가이드에는 많은 AWS SDK에 공통적인 설정, 기능 및 기타 기본 개념도 포함되어 있습니다.

이 섹션의 주제에서는 웹 브라우저에서 실행되는 SDK for JavaScript 및 Node.js용 SDK for JavaScript를 구성하는 다양한 방법을 설명합니다.

주제

- 글로벌 구성 객체 사용하기
- AWS 리전 설정
- 사용자 지정 엔드포인트 지정
- AWS를 사용한 SDK 인증
- 자격 증명 설정
- API 버전 잠금
- Node.js 고려 사항
- 브라우저 스크립트 고려 사항
- Webpack과 애플리케이션 번들링

글로벌 구성 객체 사용하기

SDK를 구성하는 방법에는 두 가지가 있습니다.

- `AWS.Config`를 사용하여 글로벌 구성을 설정합니다.
- 추가 구성 정보를 서비스 객체에 전달합니다.

AWS.Config를 사용하여 글로벌 구성을 설정하면 보통 더 쉽게 시작할 수 있지만, 서비스 수준 구성을 통해 개별 서비스를 더 효과적으로 제어할 수 있습니다. AWS.Config로 지정되는 글로벌 구성은 이후 생성하는 서비스 객체에 대한 기본 설정을 제공하여 구성을 간소화합니다. 그러나 요구 사항이 글로벌 구성과 다른 경우 개별 서비스 객체의 구성을 업데이트할 수 있습니다.

글로벌 구성 설정

코드에 aws-sdk 패키지를 로드한 후에는 AWS 전역 변수를 사용하여 SDK의 클래스에 액세스하고 개별 서비스와 상호 작용할 수 있습니다. SDK에는 글로벌 구성 객체인 AWS.Config가 포함됩니다. 이를 사용하여 애플리케이션에 필요한 SDK 구성 설정을 지정할 수 있습니다.

애플리케이션 요구 사항에 따라 AWS.Config 속성을 설정하여 SDK를 구성합니다. 다음 표에 SDK의 구성을 설정하는 데 일반적으로 사용되는 AWS.Config 속성이 요약되어 있습니다.

구성 옵션	설명
credentials	필수 서비스 및 리소스에 대한 액세스 권한을 결정하는 데 사용되는 인증 자격 증명을 지정합니다.
region	필수 서비스에 대한 요청이 이루어질 리전을 지정합니다.
maxRetries	선택 사항입니다. 지정된 요청이 재시도되는 최대 횟수를 지정합니다.
logger	선택 사항입니다. 디버깅 정보가 작성되는 로거 객체를 지정합니다.
update	선택 사항입니다. 현재 구성을 새 값으로 업데이트합니다.

구성 객체에 대한 자세한 내용은 API 참조의 [Class: AWS.Config](#) 섹션을 참조하세요.

글로벌 구성 예제

AWS.Config에서 리전과 자격 증명을 설정해야 합니다. 다음 브라우저 스크립트 예제와 같이 AWS.Config 생성자의 일부로 이러한 속성을 설정할 수 있습니다.

```
var myCredentials = new
 AWS.CognitoIdentityCredentials({IdentityPoolId:'IDENTITY_POOL_ID'});
var myConfig = new AWS.Config({
  credentials: myCredentials, region: 'us-west-2'
});
```

또한 리전을 업데이트하는 다음 예제와 같이 `AWS.Config` 메서드를 사용하여 `update`를 생성한 후 이러한 속성을 설정할 수도 있습니다.

```
myConfig = new AWS.Config();
myConfig.update({region: 'us-east-1'});
```

`getCredentials`의 정적 `AWS.config` 메서드를 호출하여 기본 자격 증명을 가져올 수 있습니다.

```
var AWS = require("aws-sdk");

AWS.config.getCredentials(function(err) {
  if (err) console.log(err.stack);
  // credentials not loaded
  else {
    console.log("Access key:", AWS.config.credentials.accessKeyId);
  }
});
```

마찬가지로, `config` 파일에서 리전을 올바르게 설정한 경우 `AWS_SDK_LOAD_CONFIG` 환경 변수를 임의의 값으로 설정하고 `AWS.config`의 정적 `region` 속성을 직접적으로 호출하여 해당 값을 가져옵니다.

```
var AWS = require("aws-sdk");

console.log("Region: ", AWS.config.region);
```

서비스별 구성 설정

SDK for JavaScript에 사용하는 각 서비스에는 해당 서비스에 대한 API의 일부인 서비스 객체를 통해 액세스합니다. 예를 들어 Amazon S3 서비스에 액세스하려면 Amazon S3 서비스 객체를 생성합니다. 해당 서비스 객체에 대한 생성자의 일부인 서비스별 구성 설정을 지정할 수 있습니다. 서비스 객체에 대한 구성 값을 설정하면 생성자가 인증 자격 증명을 포함하여 `AWS.Config`에서 사용하는 구성 값을 모두 가져옵니다.

예를 들어 여러 리전의 Amazon EC2 객체에 액세스해야 하는 경우 리전별로 Amazon EC2 서비스 객체를 생성한 다음, 그에 따라 각 서비스 객체의 리전 구성을 설정합니다.

```
var ec2_regionA = new AWS.EC2({region: 'ap-southeast-2', maxRetries: 15, apiVersion:
 '2014-10-01'});
var ec2_regionB = new AWS.EC2({region: 'us-east-1', maxRetries: 15, apiVersion:
 '2014-10-01'});
```

`AWS.Config`를 사용하여 SDK를 구성할 때 서비스별 구성 값을 설정할 수도 있습니다. 글로벌 구성 객체는 많은 서비스별 구성 옵션을 지원합니다. 서비스별 구성 파라미터에 대한 자세한 내용은 API 참조의 [`Class: AWS.Config`] 섹션을 참조하세요.

변경 불가능한 구성 데이터

글로벌 구성 변경은 새로 생성한 모든 서비스 객체에 대한 요청에 적용됩니다. 새로 생성한 서비스 객체는 먼저 현재 글로벌 구성 데이터로 구성된 후 로컬 구성 옵션으로 구성됩니다. 글로벌 `AWS.config` 객체에 수행하는 업데이트는 이전에 생성한 서비스 객체에는 적용되지 않습니다.

기존 서비스 객체는 새 구성 데이터로 수동 업데이트해야 하거나, 새 구성 데이터가 있는 새 서비스 객체를 생성 및 사용해야 합니다. 다음 예제에서는 새 구성 데이터가 있는 새 Amazon S3 서비스 객체를 생성합니다.

```
s3 = new AWS.S3(s3.config);
```

AWS 리전 설정

리전은 동일한 지리적 영역 내에서 명명된 AWS 리소스 집합입니다. 예를 들면, 미국 동부(버지니아 북부) 리전은 `us-east-1`입니다. SDK for JavaScript를 구성할 때 리전을 지정하면 SDK가 해당 리전의 리소스에 액세스할 수 있습니다. 일부 서비스는 특정 리전에서만 사용할 수 있습니다.

SDK for JavaScript는 기본적으로 리전을 선택하지 않습니다. 그러나 환경 변수, 공유 `config` 파일 또는 글로벌 구성 객체를 사용하여 리전을 설정할 수 있습니다.

클라이언트 클래스 생성자에서

서비스 객체를 인스턴스화할 때 다음과 같이 해당 리소스의 리전을 클라이언트 클래스 생성자의 일부로 지정할 수 있습니다.

```
var s3 = new AWS.S3({apiVersion: '2006-03-01', region: 'us-east-1'});
```

글로벌 구성 객체 사용하기

JavaScript 코드에서 리전을 설정하려면 다음과 같이 `AWS.Config` 글로벌 구성 객체를 업데이트해야 합니다.

```
AWS.config.update({region: 'us-east-1'});
```

현재 리전 및 각 리전에서 사용 가능한 서비스에 대한 자세한 내용은 AWS 일반 참조의 AWS 리전 및 엔드포인트 섹션을 참조하세요.

환경 변수 사용

`AWS_REGION` 환경 변수를 사용하여 리전을 설정할 수 있습니다. 이 변수를 정의하면 SDK for JavaScript가 해당 변수를 읽고 사용합니다.

공유 구성 파일 사용

공유 자격 증명 파일에 SDK에서 사용할 자격 증명을 저장할 수 있는 것과 마찬가지로 SDK에서 사용하는 `config`라는 공유 파일에 리전 및 기타 구성 설정을 보관할 수 있습니다. `AWS_SDK_LOAD_CONFIG` 환경 변수가 임의의 값으로 설정된 경우 SDK for JavaScript는 로드 시 `config` 파일을 자동으로 검색합니다. `config` 파일을 저장하는 위치는 운영 체제에 따라 다릅니다.

- Linux, macOS 또는 Unix 사용자: `~/.aws/config`
- Windows 사용자: `C:\Users\USER_NAME\.aws\config`

아직 공유 `config` 파일이 없는 경우, 지정된 디렉터리에 하나를 생성할 수 있습니다. 다음 예제의 경우 `config` 파일에서 리전과 출력 형식을 둘 다 설정합니다.

```
[default]
   region=us-east-1
   output=json
```

공유 구성 및 자격 증명 파일 사용에 대한 자세한 내용은 AWS Command Line Interface 사용 설명서의 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 또는 구성 및 자격 증명 섹션을 참조하세요.

리전 설정을 위한 우선 순위

리전 설정의 우선 순위는 다음과 같습니다.

- 어떤 리전이 클라이언트 클래스 생성자로 전달된 경우 이 리전이 사용됩니다. 그렇지 않은 경우...
- 어떤 리전이 글로벌 구성 객체에 설정된 경우 이 리전이 사용됩니다. 그렇지 않은 경우...
- `AWS_REGION` 환경 변수가 진리(truthy) 값인 경우 이 리전이 사용됩니다. 그렇지 않은 경우...
- `AMAZON_REGION` 환경 변수가 진리(truthy) 값인 경우 이 리전이 사용됩니다. 그렇지 않은 경우...
- `AWS_SDK_LOAD_CONFIG` 환경 변수가 임의의 값으로 설정되어 있고 공유 자격 증명 파일(`~/.aws/credentials` 또는 `AWS_SHARED_CREDENTIALS_FILE`에 표시된 경로)에 구성된 프로필의 리전이 포함된 경우 이 리전이 사용됩니다. 그렇지 않은 경우...
- `AWS_SDK_LOAD_CONFIG` 환경 변수가 임의의 값으로 설정되어 있고 구성 파일(`~/.aws/config` 또는 `AWS_CONFIG_FILE`에 표시된 경로)에 구성된 프로필의 리전이 포함된 경우 이 리전이 사용됩니다.

사용자 지정 엔드포인트 지정

SDK for JavaScript의 API 메서드 호출은 서비스 엔드포인트 URI로 수행됩니다. 기본적으로 이러한 엔드포인트는 코드에 대해 구성한 리전에서 빌드됩니다. 그러나 API 호출에 대해 사용자 지정 엔드포인트를 지정해야 하는 경우가 있습니다.

엔드포인트 문자열 형식

엔드포인트 값은 다음 형식의 문자열이어야 합니다.

`https://`*`{service}.{region}`*`.amazonaws.com`

ap-northeast-3 리전의 엔드포인트

일본의 `ap-northeast-3` 리전은 `EC2.describeRegions` 등의 리전 열거 API에서 반환하지 않습니다. 이 리전의 엔드포인트를 정의하려면 앞서 설명한 형식을 따르십시오. 그러면 이 리전의 Amazon EC2 엔드포인트는 다음과 같이 됩니다.

```
ec2.ap-northeast-3.amazonaws.com
```

MediaConvert용 엔드포인트

MediaConvert에서 사용할 사용자 지정 엔드포인트를 만들어야 합니다. 각 고객 계정에 사용자가 사용해야 하는 고유 엔드포인트가 할당됩니다. 다음은 MediaConvert에서 사용자 지정 엔드포인트를 사용하는 방법의 예입니다.

```
// Create MediaConvert service object using custom endpoint
```

```
var mcClient = new AWS.MediaConvert({endpoint: 'https://abcd1234.mediaconvert.us-
west-1.amazonaws.com'});

var getJobParams = {Id: 'job_ID'};

mcClient.getJob(getJobParams, function(err, data)) {
   if (err) console.log(err, err.stack); // an error occurred
   else console.log(data); // successful response
};
```

계정 API 엔드포인트를 확인하려면 API 참조의 `MediaConvert.describeEndpoints`를 참조하세요.

코드에서 사용자 지정 엔드포인트 URI의 리전과 동일한 리전을 지정해야 합니다. 리전 설정과 사용자 지정 엔드포인트 URI가 불일치하면 API 호출이 실패할 수 있습니다.

MediaConvert에 대한 자세한 내용은 API 참조의 `AWS.MediaConvert` 클래스 또는AWS Elemental MediaConvert 사용 설명서를 참조하세요.

AWS를 사용한 SDK 인증

AWS 서비스로 개발할 때는 코드가 AWS에서 인증되는 방법을 설정해야 합니다. 환경 및 사용 가능한 AWS 액세스 권한에 따라 다양한 방식으로 AWS 리소스에 대한 프로그래밍 방식의 액세스를 구성할 수 있습니다.

인증 방법을 선택하고 SDK에 맞게 구성하려면 AWS SDK 및 도구 참조 안내서의 Authentication and access를 참조하세요.

현지에서 개발 중이고 고용주로부터 인증 방법을 제공하지 않은 신규 사용자는 AWS IAM Identity Center 설정하는 것이 좋습니다. 이 방법에는 구성을 쉽게 하고 AWS 액세스 포털에 정기적으로 로그인하기 위한 AWS CLI 설치가 포함됩니다. 이 방법을 선택하는 경우 AWS SDK 및 도구 참조 안내서의 IAM Identity Center authentication 절차를 완료한 후 환경에 다음 요소가 포함되어야 합니다.

- AWS CLI는 애플리케이션을 실행하기 전에 AWS 액세스 포털 세션을 시작하는 데 사용합니다.
- SDK에서 참조할 수 있는 구성 값 세트가 포함된 `[default]` 프로필이 있는 shared AWS`config` file입니다. 이 파일의 위치를 찾으려면 AWS SDK 및 도구 참조 가이드에서 공유 파일의 위치를 참조하세요.
- 공유 `config` 파일은 `region` 설정을 지정합니다. 이는 SDK가 AWS 요청에 사용하는 기본 AWS 리전을 설정합니다. 이 리전은 사용할 리전이 지정되지 않은 SDK 서비스 요청에 사용됩니다.

- SDK는 AWS에 요청을 보내기 전에 프로필의 SSO token provider configuration을 사용하여 보안 인증을 얻습니다. `sso_role_name` 값은 IAM Identity Center 권한 집합에 연결된 IAM 역할로, 애플리케이션에서 사용되는 AWS 서비스에 대한 액세스를 허용합니다.

  다음 샘플 `config` 파일은 SSO 토큰 공급자 구성으로 설정된 기본 프로필을 보여줍니다. 프로필의 `sso_session` 설정은 이름이 지정된 `sso-session` section을 참조합니다. `sso-session` 섹션에는 AWS 액세스 포털 세션을 시작하기 위한 설정이 포함되어 있습니다.

```
[default]
sso_session = my-sso
sso_account_id = 111122223333
sso_role_name = SampleRole
region = us-east-1
output = json

[sso-session my-sso]
sso_region = us-east-1
sso_start_url = https://provided-domain.awsapps.com/start
sso_registration_scopes = sso:account:access
```

SDK for JavaScript는 IAM Identity Center 인증을 사용하기 위해 애플리케이션에 추가 패키지(예:`SSO` 및 `SSOOIDC`)를 추가할 필요가 없습니다.

AWS 액세스 포털 세션 시작

AWS 서비스에 액세스하는 애플리케이션을 실행하기 전에 SDK가 IAM Identity Center 인증을 사용하여 보안 인증을 확인하려면 활성화된 AWS 액세스 포털 세션이 있어야 합니다. 구성된 세션 길이에 따라 결국 액세스가 만료되고 SDK에 인증 오류가 발생합니다. AWS 액세스 포털에 로그인하려면 AWS CLI에서 다음 명령을 실행합니다.

```
aws sso login
```

지침에 따라 기본 프로필을 설정했다면 `--profile` 옵션으로 명령을 직접적으로 호출할 필요가 없습니다. SSO 토큰 공급자 구성에서 명명된 프로필을 사용하는 경우 `aws sso login --profile named-profile` 명령을 사용합니다.

필요할 경우 이미 활성 세션이 있는지 테스트하려면 다음 AWS CLI 명령을 실행합니다.

```
aws sts get-caller-identity
```

세션이 활성 상태인 경우 이 명령에 대한 응답은 공유 `config` 파일에 구성된 IAM Identity Center 계정 및 권한 집합을 보고합니다.

> **Note**
>
> 이미 활성 AWS 액세스 포털 세션이 있고 `aws sso login`을 실행하는 경우 보안 인증 정보를 제공하지 않아도 됩니다.
> 로그인 과정에서 데이터 AWS CLI 액세스를 허용하라는 메시지가 표시될 수 있습니다. AWS CLI는 SDK for Python를 기반으로 구축되므로 권한 메시지에는 `botocore` 이름의 변형이 포함될 수 있습니다.

세부 인증 정보

인간 사용자(인간 ID라고도 함)는 애플리케이션의 사용자, 관리자, 개발자, 운영자 및 소비자입니다. AWS 환경 및 애플리케이션에 액세스하려면 ID가 있어야 합니다. 조직의 구성원인 인간 사용자, 즉 개발자는 작업 인력 ID라고도 합니다.

AWS에 액세스할 때 임시 보안 인증을 사용합니다. 임시 보안 인증을 제공하는 역할을 가정하여 인간 사용자가 AWS 계정에 페더레이션 액세스를 제공하도록 ID 공급자를 사용할 수 있습니다. 중앙 액세스 관리를 위해 AWS IAM Identity Center(IAM Identity Center)을 사용하여 계정에 대한 액세스 권한과 해당 계정 내 권한을 관리하는 것이 좋습니다. 더 많은 대안을 보려면 다음을 참조하세요.

- 모범 사례에 대해 자세히 알아보려면 IAM 사용 설명서에서 IAM의 보안 모범 사례를 참조하세요.
- 단기 AWS 보안 인증 정보를 만들려면 IAM 사용 설명서에서 임시 보안 인증 정보를 참조하세요.
- 다른 보안 인증 공급자에 대해 알아보려면 AWS SDK 및 도구 참조 가이드에서 표준화된 자격 증명 공급자를 참조하세요.

자격 증명 설정

AWS는 보안 인증을 사용하여 누가 서비스를 호출하는지와 요청한 리소스에 대한 액세스가 허용되는지 여부를 확인합니다.

웹 브라우저에서 실행되든 Node.js 서버에서 실행되든, JavaScript 코드가 API를 통해 서비스에 액세스하려면 먼저 유효한 인증 자격 증명을 얻어야 합니다. `AWS.Config`를 사용하거나 서비스별로 서비스 객체에 인증 자격 증명을 직접 전달하여 구성 객체에서 인증 자격 증명을 전역으로 설정할 수 있습니다.

웹 브라우저에서 Node.js와 JavaScript 간에 서로 다른 인증 자격 증명을 설정하는 방법에는 여러 가지가 있습니다. 이 섹션의 주제에서는 Node.js 또는 웹 브라우저에서 인증 자격 증명을 설정하는 방법을 설명합니다. 각각의 경우에 옵션은 권장 순서로 제공됩니다.

인증 자격 증명에 대한 모범 사례

인증 자격 증명을 올바르게 설정하면 애플리케이션 또는 브라우저 스크립트가 필요한 서비스 및 리소스에 액세스할 수 있는 동시에 미션 크리티컬 애플리케이션에 미치거나 중요한 데이터를 손상시킬 수 있는 보안 문제에 대한 노출을 최소화할 수 있습니다.

인증 자격 증명을 설정할 때 적용되는 중요한 원칙은 항상 작업에 필요한 최소 권한을 부여하는 것입니다. 최소 권한을 초과하는 권한을 제공하고 그 결과로 추후 발견할 수 있는 보안 문제를 해결하기 보다는, 리소스에 대한 최소 권한을 제공하고 필요에 따라 권한을 추가하는 것이 더 안전합니다. 예를 들어 Amazon S3 버킷 또는 DynamoDB 테이블의 객체 같은 개별 리소스를 읽고 쓸 필요가 있지 않은 한, 그러한 권한을 읽기 전용으로 설정합니다.

최소 권한 부여에 관한 자세한 내용은 IAM 사용 설명서의 모범 사례 주제에서 최소 권한 적용 섹션을 참조하세요.

> **Warning**
>
> 애플리케이션 또는 브라우저 스크립트 내부에서 인증 자격 증명을 하드 코딩할 수 있더라도 그렇게 하지 않는 것이 좋습니다. 자격 증명을 하드 코딩하면 민감한 정보가 노출될 위험이 있습니다.

액세스 키 관리에 대한 자세한 정보는 AWS 일반 참조에서 AWS 액세스 키 관리 모범 사례를 참조하세요.

주제

- Node.js에서 자격 증명 설정
- 웹 브라우저에서 자격 증명 설정

Node.js에서 자격 증명 설정

Node.js에서 SDK에 인증 자격 증명을 제공하는 방법에는 여러 가지가 있습니다. 그 가운데는 더 안전한 방법도 있고 애플리케이션 개발 시에 더 편리한 방법도 있습니다. Node.js에서 인증 자격 증명을 얻

을 때 로드하는 JSON 파일 및 환경 변수 등 둘 이상의 소스를 사용하는 경우 주의해야 합니다. 변경 발생에 대한 인식 없이 코드가 실행되는 권한을 변경할 수 있습니다.

다음은 권장 순서로 인증 자격 증명을 제공할 수 있는 방법입니다.

1. AWS Identity and Access Management(IAM)에서 Amazon EC2용 역할 로드
2. 공유 인증 자격 증명 파일(`~/.aws/credentials`)에서 로드
3. 환경 변수에서 로드
4. 디스크의 JSON 파일에서 로드
5. JavaScript SDK에서 제공하는 기타 자격 증명 공급자 클래스

SDK에서 둘 이상의 자격 증명 소스를 사용할 수 있는 경우 기본 선택 우선 순위는 다음과 같습니다.

1. 서비스 클라이언트 생성자를 통해 명시적으로 설정된 자격 증명
2. 환경 변수
3. 공유 자격 증명 파일
4. ECS 자격 증명 공급자에서 로드된 자격 증명(해당되는 경우)
5. 공유 AWS config 파일 또는 공유 자격 증명 파일에서 지정된 자격 증명 프로세스를 사용해 얻은 자격 증명. 자세한 내용은 the section called “구성된 자격 증명 프로세스를 사용하는 자격 증명” 단원을 참조하십시오.
6. Amazon EC2 인스턴스의 자격 증명 공급자를 사용해 AWS IAM 에서 로드된 자격 증명(인스턴스 메타데이터에서 구성된 경우)

자세한 내용은 API 참조의 `Class: AWS.Credentials` 및 `Class: AWS.CredentialProviderChain` 섹션을 참조하세요.

Warning

애플리케이션에서 AWS 자격 증명을 하드 코딩할 수 있더라도 그렇게 하지 않는 것이 좋습니다. 인증 자격 증명을 하드 코딩하면 액세스 키 ID 및 보안 액세스 키가 노출될 위험이 있습니다.

이 섹션의 주제에서는 인증 자격 증명을 Node.js로 로드하는 방법을 설명합니다.

주제

- Amazon EC2의 IAM 역할에서 Node.js의 자격 증명 로드
- Node.js Lambda 함수의 인증 자격 증명 로드
- 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드
- 환경 변수에서 Node.js에 인증 자격 증명 로드
- JSON 파일에서 Node.js에 인증 자격 증명 로드
- 구성된 자격 증명 프로세스를 사용해 Node.js에서 자격 증명 로딩

Amazon EC2의 IAM 역할에서 Node.js의 자격 증명 로드

Amazon EC2 인스턴스에서 Node.js 애플리케이션을 실행하는 경우 Amazon EC2의 IAM 역할을 활용하여 해당 인스턴스에 보안 인증을 자동으로 제공할 수 있습니다. IAM 역할을 사용하도록 인스턴스를 구성하면 SDK가 애플리케이션의 IAM 자격 증명을 자동으로 선택하므로 자격 증명을 수동으로 제공할 필요가 없습니다.

Amazon EC2 인스턴스에 IAM 역할을 추가하는 방법에 대한 자세한 내용은 AWS SDK 및 도구 참조 안내서의 Amazon EC2 인스턴스용 IAM 역할 사용을 참조하세요.

Node.js Lambda 함수의 인증 자격 증명 로드

AWS Lambda 함수를 만들 때 이 함수를 실행할 권한이 있는 특별한 IAM 역할을 만들어야 합니다. 이 역할을 실행 역할이라고 합니다. Lambda 함수를 설정할 때 해당 실행 역할로 생성한 IAM 역할을 지정해야 합니다.

실행 역할은 Lambda 함수에 다른 웹 서비스를 실행 및 간접 호출하는 데 필요한 자격 증명을 제공합니다. 따라서 Lambda 함수 내에 쓰는 Node.js 코드에 인증 자격 증명을 제공할 필요가 없습니다.

Lambda 실행 역할 생성에 관한 자세한 내용은 AWS Lambda 개발자 안내서의 권한 관리: IAM 역할(실행 역할) 활용 섹션을 참조하세요.

공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드

SDK 및 명령줄 인터페이스에서 사용하는 공유 파일에 AWS 자격 증명 데이터를 보관할 수 있습니다. SDK for JavaScript가 로드되면 "credentials"라는 공유 자격 증명 파일이 자동으로 검색됩니다. 공유 인증 자격 증명 파일을 보관하는 위치는 운영 체제에 따라 다릅니다.

- Linux, Unix 및 macOS의 공유 자격 증명 파일: `~/.aws/credentials`
- Windows의 공유 자격 증명 파일: `C:\Users\USER_NAME\.aws\credentials`

공유된 자격 증명 파일이 아직 없는 경우 AWS를 사용한 SDK 인증를 참조하세요. 이 지침을 따르면 자격 증명 파일에 다음과 유사한 텍스트가 표시되어야 합니다. 이 텍스트에서 *<YOUR_ACCESS_KEY_ID>*는 액세스 키 ID이고 *<YOUR_SECRET_ACCESS_KEY>*는 보안 액세스 키입니다.

```
[default]
aws_access_key_id = <YOUR_ACCESS_KEY_ID>
aws_secret_access_key = <YOUR_SECRET_ACCESS_KEY>
```

사용 중인 이 파일을 보여주는 예제는 Node.js에서 시작하기를 참조하세요.

`[default]` 섹션 머리글은 인증 자격 증명의 기본 프로필 및 관련 값을 지정합니다. 동일한 공유 구성 파일에 각각 고유한 인증 자격 증명 정보가 있는 추가 프로필을 만들 수 있습니다. 다음 예제에서는 기본 프로필과 추가 프로필 두 개가 있는 구성 파일을 보여 줍니다.

```
[default] ; default profile
aws_access_key_id = <DEFAULT_ACCESS_KEY_ID>
aws_secret_access_key = <DEFAULT_SECRET_ACCESS_KEY>

[personal-account] ; personal account profile
aws_access_key_id = <PERSONAL_ACCESS_KEY_ID>
aws_secret_access_key = <PERSONAL_SECRET_ACCESS_KEY>

[work-account] ; work account profile
aws_access_key_id = <WORK_ACCESS_KEY_ID>
aws_secret_access_key = <WORK_SECRET_ACCESS_KEY>
```

기본적으로 SDK는 `AWS_PROFILE` 환경 변수를 확인하여 사용할 프로필을 결정합니다. 환경에서 `AWS_PROFILE` 변수가 설정되지 않은 경우, SDK는 `[default]` 프로필의 인증 자격 증명을 사용합니다. 대체 프로필 중 하나를 사용하려면 `AWS_PROFILE` 환경 변수의 값을 설정하거나 변경합니다. 예를 들어 위에 표시된 구성 파일을 통해 작업 계정의 자격 증명을 사용하려면 `AWS_PROFILE` 환경 변수를 `work-account`(운영 체제에 적합한 경우)로 설정합니다.

> **Note**
>
> 환경 변수를 설정할 경우에는 셸 또는 명령 환경에서 변수를 사용할 수 있도록 (운영 체제의 필요에 따라) 나중에 적절한 조치를 취해야 합니다.

환경 변수를 설정한 후(필요한 경우) SDK를 사용하는 JavaScript 파일(예: `script.js` 파일)을 실행할 수 있습니다.

```
$ node script.js
```

다음 예제와 같이 인증 자격 증명 공급자를 선택하거나 SDK를 로드하기 전에 `process.env.AWS_PROFILE`을 설정하여 SDK에서 사용하는 프로필을 명시적으로 선택할 수도 있습니다.

```
var credentials = new AWS.SharedIniFileCredentials({profile: 'work-account'});
AWS.config.credentials = credentials;
```

환경 변수에서 Node.js에 인증 자격 증명 로드

SDK는 환경에서 변수로 설정된 AWS 자격 증명을 자동으로 감지하고 SDK 요청에 이 인증 자격 증명을 사용하므로 애플리케이션에서 자격 증명을 관리할 필요가 없습니다. 인증 자격 증명을 제공하기 위해 설정하는 환경 변수는 다음과 같습니다.

- `AWS_ACCESS_KEY_ID`
- `AWS_SECRET_ACCESS_KEY`
- `AWS_SESSION_TOKEN`

환경 변수를 설정하는 방법에 대한 자세한 내용은 AWS SDK 및 도구 참조 가이드의 환경 변수 지원 섹션을 참조하세요.

JSON 파일에서 Node.js에 인증 자격 증명 로드

`AWS.config.loadFromPath`를 사용하여 JSON 문서에서 디스크에 구성 및 인증 자격 증명을 로드할 수 있습니다. 지정된 경로는 프로세스의 현재 작업 디렉터리와 관련이 있습니다. 예를 들어 다음 내용이 있는 `'config.json'` 파일에서 인증 자격 증명을 로드하려면:

```
{ "accessKeyId": <YOUR_ACCESS_KEY_ID>, "secretAccessKey": <YOUR_SECRET_ACCESS_KEY>,
 "region": "us-east-1" }
```

다음 코드를 사용합니다.

```
var AWS = require("aws-sdk");
AWS.config.loadFromPath('./config.json');
```

Note

JSON 문서에서 구성 데이터를 로드하면 기존 구성 데이터가 모두 재설정됩니다. 이 기술을 사용한 후 추가 구성 데이터를 추가합니다. JSON 문서에서 인증 자격 증명을 로드하는 것은 브라우저 스크립트에서 지원되지 않습니다.

구성된 자격 증명 프로세스를 사용해 Node.js에서 자격 증명 로딩

SDK에 없는 메소드를 사용해 자격 증명을 공급받을 수 있습니다. 이를 위해서는 먼저 공유 AWS config 파일 또는 공유 자격 증명 파일에서 자격 증명 프로세스를 지정합니다. `AWS_SDK_LOAD_CONFIG` 환경 변수가 임의의 값으로 설정되어 있으면 SDK가 자격 증명 파일(있는 경우)에 지정된 프로세스보다 config 파일에 지정된 프로세스를 선호합니다.

공유 AWS 구성 파일 또는 공유 자격 증명 파일에서 자격 증명 프로세스를 지정하는 방법에 대한 자세한 내용은 AWS CLI 명령 참조서 중 외부 프로세스에서 자격 증명 소싱의 내용을 참조하세요.

`AWS_SDK_LOAD_CONFIG` 환경 변수의 사용 방법에 대한 자세한 내용은 본 문서에서 the section called "공유 구성 파일 사용" 단원을 참조하세요.

웹 브라우저에서 자격 증명 설정

브라우저 스크립트에서 SDK에 인증 자격 증명을 제공하는 방법에는 여러 가지가 있습니다. 그 가운데는 더 안전한 방법도 있고 스크립트 개발 시에 더 편리한 방법도 있습니다. 다음은 권장 순서로 인증 자격 증명을 제공할 수 있는 방법입니다.

1. Amazon Cognito 자격 증명을 사용하여 사용자를 인증하고 자격 증명 제공
2. 웹 연동 자격 증명 사용
3. 스크립트에서 하드 코딩

Warning

스크립트에서 AWS 자격 증명을 하드 코딩하지 않는 것이 좋습니다. 인증 자격 증명을 하드 코딩하면 액세스 키 ID 및 보안 액세스 키가 노출될 위험이 있습니다.

주제

- Amazon Cognito 자격 증명을 사용하여 사용자 인증
- 웹 연동 자격 증명을 사용하여 사용자 인증
- 웹 연동 자격 증명 예제

Amazon Cognito 자격 증명을 사용하여 사용자 인증

브라우저 스크립트에 대한 AWS 보안 인증을 얻는 권장 방법은 Amazon Cognito 자격 증명 보안 인증 객체인 `AWS.CognitoIdentityCredentials`를 사용하는 것입니다. Amazon Cognito를 사용하면 타사 자격 증명 공급자를 통해 사용자를 인증할 수 있습니다.

Amazon Cognito 자격 증명을 사용하려면 먼저, Amazon Cognito 콘솔에서 자격 증명 풀을 생성해야 합니다. 자격 증명 풀은 애플리케이션이 사용자에게 제공하는 자격 증명 그룹을 나타냅니다. 사용자에게 제공되는 자격 증명은 각 사용자 계정을 고유하게 식별합니다. Amazon Cognito 자격 증명(identity)은 자격 증명(credential)이 아닙니다. AWS Security Token Service(AWS STS)에서 웹 자격 증명 연동 지원을 사용하여 인증 자격 증명으로 교환됩니다.

Amazon Cognito는 `AWS.CognitoIdentityCredentials` 객체를 사용하여 여러 자격 증명 공급자의 자격 증명 추상화를 관리할 수 있습니다. 그러면 로드되는 자격 증명이 AWS STS의 인증 자격 증명과 교환됩니다.

Amazon Cognito 자격 증명 보안 인증 객체 구성

아직 자격 증명 풀을 생성하지 않은 경우 `AWS.CognitoIdentityCredentials`를 구성하기 전에 Amazon Cognito 콘솔에서 브라우저 스크립트와 함께 사용할 자격 증명 풀을 생성합니다. 자격 증명 풀에 대한 인증 IAM 역할과 미인증 IAM 역할을 모두 생성하고 연결합니다.

미인증 사용자는 자격 증명이 인증되지 않았으므로 앱 혹은 자격 증명의 인증이 중요하지 않은 경우 게스트 사용자 역할에 적합합니다. 인증받은 사용자는 자격 증명을 확인하는 타사 자격 증명 공급자를 통해 애플리케이션에 로그인합니다. 미인증 사용자의 액세스 권한을 허용하지 않도록 리소스 권한 범위를 충분히 정했는지 확인하세요.

연결된 자격 증명 공급자로 자격 증명 풀을 구성한 후 `AWS.CognitoIdentityCredentials`를 사용하여 사용자를 인증할 수 있습니다. `AWS.CognitoIdentityCredentials`를 사용하도록 애플리케이션 자격 증명을 구성하려면, `credentials` 또는 서비스당 구성의 `AWS.Config` 속성을 설정하세요. 다음 예에는 `AWS.Config`가 사용됩니다.

```
AWS.config.credentials = new AWS.CognitoIdentityCredentials({
  IdentityPoolId: 'us-east-1:1699ebc0-7900-4099-b910-2df94f52a030',
  Logins: { // optional tokens, used for authenticated login
```

```
    'graph.facebook.com': 'FBTOKEN',
    'www.amazon.com': 'AMAZONTOKEN',
    'accounts.google.com': 'GOOGLETOKEN'
  }
});
```

선택 사항인 Logins 속성은 공급자의 자격 증명 토큰에 대한 자격 증명 공급자 이름의 맵입니다. 자격 증명 공급자에게서 토큰을 받는 방법은 어떤 공급자를 사용하느냐에 따라 다릅니다. 예를 들어 Facebook이 자격 증명 공급자 중 하나인 경우 [Facebook SDK](https://developers.facebook.com/docs/facebook-login/web)의 FB.login 함수를 사용하여 자격 증명 공급자 토큰을 얻을 수 있습니다.

```
FB.login(function (response) {
  if (response.authResponse) { // logged in
    AWS.config.credentials = new AWS.CognitoIdentityCredentials({
      IdentityPoolId: 'us-east-1:1699ebc0-7900-4099-b910-2df94f52a030',
      Logins: {
        'graph.facebook.com': response.authResponse.accessToken
      }
    });

    s3 = new AWS.S3; // we can now create our service object

    console.log('You are now logged in.');
  } else {
    console.log('There was a problem logging you in.');
  }
});
```

인증되지 않은 사용자를 인증된 사용자로 전환

Amazon Cognito는 인증된 사용자와 인증되지 않은 사용자를 모두 지원합니다. 인증되지 않은 사용자는 자격 증명 공급자로 로그인하지 않았더라도 리소스에 대한 액세스 권한을 받습니다. 이 액세스 권한 등급은 로그인하기 전에 사용자에게 콘텐츠를 표시하는 데 유용합니다. 각 미인증 사용자는 개별적으로 로그인되지 않고 인증되지 않았더라도 Amazon Cognito에 고유한 자격 증명이 있습니다.

처음에 인증되지 않은 사용자

사용자는 일반적으로 Logins 속성 없이 구성 객체의 인증 자격 증명 속성을 설정한 인증되지 않은 역할로 시작합니다. 이 경우, 기본 구성은 다음과 같을 수 있습니다.

```
// set the default config object
```

```
var creds = new AWS.CognitoIdentityCredentials({
 IdentityPoolId: 'us-east-1:1699ebc0-7900-4099-b910-2df94f52a030'
});
AWS.config.credentials = creds;
```

인증된 사용자로 전환

인증되지 않은 사용자가 자격 증명 공급자에 로그인한 상태에서 현재 사용자가 토큰을 갖고 있다면, 인증 자격 증명 객체를 업데이트하고 `Logins` 토큰을 추가하는 사용자 지정 함수를 호출하여 인증되지 않은 사용자를 인증된 사용자로 전환할 수 있습니다.

```
// Called when an identity provider has a token for a logged in user
function userLoggedIn(providerName, token) {
  creds.params.Logins = creds.params.Logins || {};
  creds.params.Logins[providerName] = token;

  // Expire credentials to refresh them on the next request
  creds.expired = true;
}
```

또한 `CognitoIdentityCredentials` 객체를 생성할 수 있습니다. 이 경우 생성한 기존 서비스 객체의 인증 자격 증명 속성을 재설정해야 합니다. 서비스 객체는 객체 초기화 시 글로벌 구성에서만 읽습니다.

`CognitoIdentityCredentials` 객체에 대한 자세한 내용은 AWS SDK for JavaScript API 참조의 [`AWS.CognitoIdentityCredentials`] 섹션을 참조하세요.

웹 연동 자격 증명을 사용하여 사용자 인증

웹 ID 페더레이션을 사용하여 AWS 리소스에 액세스하도록 개별 자격 증명 공급자를 직접 구성할 수 있습니다. AWS는 현재 여러 자격 증명 공급자를 통해 웹 ID 페더레이션을 사용한 사용자 인증을 지원합니다.

- Login with Amazon
- Facebook 로그인
- Google 로그인

먼저 애플리케이션이 지원하는 공급자에 애플리케이션을 등록해야 합니다. 다음으로 IAM 역할을 생성하고 이 역할의 권한을 설정합니다. 그런 다음 생성한 IAM 역할을 사용하여 각 자격 증명 공급자를 통

해 이 역할에 대해 구성한 권한을 부여합니다. 예를 들어 Facebook을 통해 로그인한 사용자가 관리 중인 특정 Amazon S3 버킷에 대한 읽기 액세스 권한을 갖도록 허용하는 역할을 설정할 수 있습니다.

구성된 권한이 있는 IAM 역할과 선택한 자격 증명 공급자에 등록된 애플리케이션이 모두 있으면, 다음과 같이 헬퍼 코드를 사용하여 IAM 역할의 인증 자격 증명을 얻도록 SDK를 설정할 수 있습니다.

```
AWS.config.credentials = new AWS.WebIdentityCredentials({
   RoleArn: 'arn:aws:iam::<AWS_ACCOUNT_ID>/:role/<WEB_IDENTITY_ROLE_NAME>',
   ProviderId: 'graph.facebook.com|www.amazon.com', // this is null for Google
   WebIdentityToken: ACCESS_TOKEN
});
```

`ProviderId` 파라미터의 값은 지정한 자격 증명 공급자에 따라 다릅니다. `WebIdentityToken` 파라미터의 값은 해당 자격 증명 공급자에 로그인하여 검색한 액세스 토큰입니다. 각 자격 증명 공급자의 액세스 토큰을 구성 및 검색하는 방법에 대한 자세한 내용은 해당 자격 증명 공급자의 설명서를 참조하세요.

1단계: 자격 증명 공급자에 등록

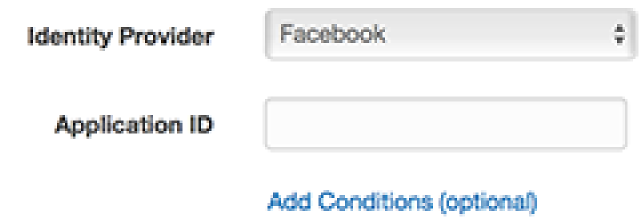
시작하려면 지원하기로 선택한 자격 증명 공급자에 애플리케이션을 등록합니다. 애플리케이션과 애플리케이션 작성자를 식별하는 정보를 제공하라는 메시지가 표시됩니다. 이를 통해 자격 증명 공급자는 사용자 정보를 받는 사람을 알 수 있습니다. 각각의 경우에 자격 증명 공급자는 사용자 역할을 구성하는 데 사용하는 애플리케이션 ID를 발급합니다.

2단계: 자격 증명 공급자의 IAM 역할 생성

자격 증명 공급자로부터 애플리케이션 ID를 얻은 후, https://console.aws.amazon.com/iam/에서 IAM 콘솔로 이동하여 새 IAM 역할을 생성합니다.

자격 증명 공급자의 IAM 역할을 생성하는 방법

1. 콘솔의 역할 섹션으로 이동한 후 Create New Role(새 역할 생성)을 선택합니다.
2. 사용을 추적하는 데 도움이 되는 새 역할의 이름을 입력한 후(예: **`facebookIdentity`**) Next Step(다음 단계)를 선택합니다.
3. Select Role Type(역할 유형 선택)에서 Role for Identity Provider Access(자격 증명 공급자 액세스 역할)를 선택합니다.
4. Grant access to web identity providers(웹 자격 증명 공급자에게 액세스 부여)에서 Select(선택)를 선택합니다.
5. 자격 증명 공급자 목록에서 이 IAM 역할에 사용할 자격 증명 공급자를 선택합니다.



6. 애플리케이션 ID에 자격 증명 공급자가 제공한 애플리케이션 ID를 입력한 후 Next Step(다음 단계)를 선택합니다.
7. 표시하려는 리소스에 대한 권한을 구성하여 특정 리소스의 특정 작업에 대한 액세스를 허용합니다. IAM 권한에 대한 자세한 내용은 IAM 사용 설명서의 AWS IAM 권한 개요를 참조하세요. 역할의 신뢰 관계를 검토하고 필요한 경우 사용자 지정한 후, Next Step(다음 단계)를 선택합니다.
8. 필요한 추가 정책을 연결한 후 Next Step(다음 단계)를 선택합니다. IAM 정책 작성에 대한 자세한 내용은 IAM 사용 설명서의 IAM 정책 개요를 참조하세요.
9. 새 역할을 검토한 후 Create Role(역할 생성)을 선택합니다.

역할에 여러 제약 조건을 제공할 수 있습니다(예: 특정 사용자 ID로 범위 지정). 역할이 리소스에 대한 쓰기 권한을 부여하는 경우, 역할의 범위를 올바른 권한이 있는 사용자로 올바르게 지정해야 합니다. 그렇지 않으면 Amazon, Facebook 또는 Google ID를 가진 모든 사용자가 애플리케이션의 리소스를 수정할 수 있습니다.

웹 ID 페더레이션에 대한 자세한 내용은 IAM 사용 설명서의 웹 ID 페더레이션 정보를 참조하세요.

3단계: 로그인 후 공급자 액세스 토큰 얻기

자격 증명 공급자의 SDK를 사용하여 애플리케이션의 로그인 작업을 설정합니다. OAuth 또는 OpenID를 사용하여 사용자 로그인을 지원하는 자격 증명 공급자에서 JavaScript SDK를 다운로드하고 설치할 수 있습니다. 애플리케이션에 SDK 코드를 다운로드하고 설정하는 방법은 해당 자격 증명 공급자의 SDK 설명서를 참조하세요.

- Login with Amazon
- Facebook 로그인
- Google 로그인

4단계: 임시 인증 자격 증명 얻기

애플리케이션, 역할 및 리소스 권한을 구성한 후, 애플리케이션에 코드를 추가하여 임시 인증 자격 증명을 얻습니다. 이러한 인증 자격 증명은 웹 자격 증명 연동을 사용하여 AWS Security Token Service

를 통해 제공됩니다. 사용자가 자격 증명 공급자에 로그인하면 액세스 토큰이 반환됩니다. 이 자격 증명 공급자에 대해 생성한 IAM 역할의 ARN을 사용하여 `AWS.WebIdentityCredentials` 객체를 설정합니다.

```
AWS.config.credentials = new AWS.WebIdentityCredentials({
    RoleArn: 'arn:aws:iam::<AWS_ACCOUNT_ID>:role/<WEB_IDENTITY_ROLE_NAME>',
    ProviderId: 'graph.facebook.com|www.amazon.com', // Omit this for Google
    WebIdentityToken: ACCESS_TOKEN // Access token from identity provider
});
```

이후 생성되는 서비스 객체에 적절한 인증 자격 증명이 생깁니다. `AWS.config.credentials` 속성을 설정하기 전에 생성된 객체에는 현재 인증 자격 증명이 없습니다.

액세스 토큰을 검색하기 전에 `AWS.WebIdentityCredentials`를 생성할 수도 있습니다. 이렇게 하면 액세스 토큰을 로드하기 전에 인증 자격 증명을 사용하는 서비스 객체를 생성할 수 있습니다. 이를 수행하려면 `WebIdentityToken` 파라미터 없이 인증 자격 증명 객체를 생성합니다.

```
AWS.config.credentials = new AWS.WebIdentityCredentials({
  RoleArn: 'arn:aws:iam::<AWS_ACCOUNT_ID>:role/<WEB_IDENTITY_ROLE_NAME>',
  ProviderId: 'graph.facebook.com|www.amazon.com' // Omit this for Google
});

// Create a service object
var s3 = new AWS.S3;
```

그런 다음 액세스 토큰이 포함된 자격 증명 공급자 SDK의 콜백에서 `WebIdentityToken`을 설정합니다.

```
AWS.config.credentials.params.WebIdentityToken = accessToken;
```

웹 연동 자격 증명 예제

다음은 웹 연동 자격 증명을 사용하여 브라우저 JavaScript에서 인증 자격 증명을 얻는 몇 가지 예제입니다. 이러한 예제는 자격 증명 공급자가 애플리케이션으로 리디렉션할 수 있도록 http:// 또는 https:// 호스트 체계에서 실행해야 합니다.

Login with Amazon 예제

다음 코드에서는 Login with Amazon을 자격 증명 공급자로 사용하는 방법을 보여 줍니다.

```
<a href="#" id="login">
```

```
  <img border="0" alt="Login with Amazon"
    src="https://images-na.ssl-images-amazon.com/images/G/01/lwa/
btnLWA_gold_156x32.png"
    width="156" height="32" />
</a>
<div id="amazon-root"></div>
<script type="text/javascript">
  var s3 = null;
  var clientId = 'amzn1.application-oa2-client.1234567890abcdef'; // client ID
  var roleArn = 'arn:aws:iam::<AWS_ACCOUNT_ID>:role/<WEB_IDENTITY_ROLE_NAME>';

  window.onAmazonLoginReady = function() {
    amazon.Login.setClientId(clientId); // set client ID

    document.getElementById('login').onclick = function() {
      amazon.Login.authorize({scope: 'profile'}, function(response) {
        if (!response.error) { // logged in
          AWS.config.credentials = new AWS.WebIdentityCredentials({
            RoleArn: roleArn,
            ProviderId: 'www.amazon.com',
            WebIdentityToken: response.access_token
          });

          s3 = new AWS.S3();

          console.log('You are now logged in.');
        } else {
          console.log('There was a problem logging you in.');
        }
      });
    };
  };

  (function(d) {
    var a = d.createElement('script'); a.type = 'text/javascript';
    a.async = true; a.id = 'amazon-login-sdk';
    a.src = 'https://api-cdn.amazon.com/sdk/login1.js';
    d.getElementById('amazon-root').appendChild(a);
  })(document);
</script>
```

Facebook 로그인 예제

다음 코드에서는 Facebook 로그인을 자격 증명 공급자로 사용하는 방법을 보여 줍니다.

```
<button id="login">Login</button>
<div id="fb-root"></div>
<script type="text/javascript">
var s3 = null;
var appId = '1234567890'; // Facebook app ID
var roleArn = 'arn:aws:iam::<AWS_ACCOUNT_ID>:role/<WEB_IDENTITY_ROLE_NAME>';

window.fbAsyncInit = function() {
  // init the FB JS SDK
  FB.init({appId: appId});

  document.getElementById('login').onclick = function() {
    FB.login(function (response) {
      if (response.authResponse) { // logged in
        AWS.config.credentials = new AWS.WebIdentityCredentials({
          RoleArn: roleArn,
          ProviderId: 'graph.facebook.com',
          WebIdentityToken: response.authResponse.accessToken
        });

        s3 = new AWS.S3;

        console.log('You are now logged in.');
      } else {
        console.log('There was a problem logging you in.');
      }
    });
  };
};

// Load the FB JS SDK asynchronously
(function(d, s, id){
   var js, fjs = d.getElementsByTagName(s)[0];
   if (d.getElementById(id)) {return;}
   js = d.createElement(s); js.id = id;
   js.src = "//connect.facebook.net/en_US/all.js";
   fjs.parentNode.insertBefore(js, fjs);
 }(document, 'script', 'facebook-jssdk'));
</script>
```

Google+ 로그인 예제

다음 코드에서는 Google+ 로그인을 자격 증명 공급자로 사용하는 방법을 보여 줍니다. Google의 웹 자격 증명 연동에 사용되는 액세스 토큰은 다른 자격 증명 공급자와 마찬가지로 `access_token` 대신에 `response.id_token`에 저장됩니다.

```
<span
  id="login"
  class="g-signin"
  data-height="short"
  data-callback="loginToGoogle"
  data-cookiepolicy="single_host_origin"
  data-requestvisibleactions="http://schemas.google.com/AddActivity"
  data-scope="https://www.googleapis.com/auth/plus.login">
</span>
<script type="text/javascript">
  var s3 = null;
  var clientID = '1234567890.apps.googleusercontent.com'; // Google client ID
  var roleArn = 'arn:aws:iam::<AWS_ACCOUNT_ID>:role/<WEB_IDENTITY_ROLE_NAME>';

  document.getElementById('login').setAttribute('data-clientid', clientID);
  function loginToGoogle(response) {
    if (!response.error) {
      AWS.config.credentials = new AWS.WebIdentityCredentials({
        RoleArn: roleArn, WebIdentityToken: response.id_token
      });

      s3 = new AWS.S3();

      console.log('You are now logged in.');
    } else {
      console.log('There was a problem logging you in.');
    }
  }

  (function() {
    var po = document.createElement('script'); po.type = 'text/javascript'; po.async =
 true;
    po.src = 'https://apis.google.com/js/client:plusone.js';
    var s = document.getElementsByTagName('script')[0]; s.parentNode.insertBefore(po,
 s);
  })();
 </script>
```

API 버전 잠금

AWS 서비스에는 API 호환성을 추적하는 API 버전 번호가 있습니다. AWS 서비스의 API 버전은 `YYYY-mm-dd` 형식의 날짜 문자열로 식별됩니다. 예를 들어 Amazon S3의 현재 API 버전은 `2006-03-01`입니다.

프로덕션 코드에서 서비스의 API 버전을 사용하는 경우 잠그는 것이 좋습니다. 그러면 SDK 업데이트로 인한 서비스 변경으로부터 애플리케이션을 격리할 수 있습니다. 서비스 객체를 생성할 때 API 버전을 지정하지 않으면 SDK가 기본적으로 최신 API 버전을 사용합니다. 그러면 애플리케이션이 애플리케이션에 부정적인 영향을 주는 변경 사항이 포함된 업데이트된 API를 참조할 수 있습니다.

서비스에 사용하는 API 버전을 잠그려면 서비스 객체를 생성할 때 `apiVersion` 파라미터를 전달합니다. 다음 예제에서는 새로 생성된 `AWS.DynamoDB` 서비스 객체가 `2011-12-05` API 버전으로 잠깁니다.

```
var dynamodb = new AWS.DynamoDB({apiVersion: '2011-12-05'});
```

`apiVersions`에서 `AWS.Config` 파라미터를 지정하여 서비스 API 버전 집합을 전역으로 구성할 수 있습니다. 예를 들어 현재 Amazon Redshift API와 함께 특정 버전의 DynamoDB 및 Amazon EC2 API를 설정하려면 다음과 같이 `apiVersions`를 설정합니다.

```
AWS.config.apiVersions = {
  dynamodb: '2011-12-05',
  ec2: '2013-02-01',
  redshift: 'latest'
};
```

API 버전 가져오기

서비스의 API 버전을 가져오려면 서비스의 참조 페이지(예: Amazon S3용 https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/S3.html)에서 API 버전 잠금 섹션을 참조하세요.

Node.js 고려 사항

Node.js 코드는 JavaScript이지만 Node.js에서 AWS SDK for JavaScript를 사용하는 것은 브라우저 스크립트에서 SDK를 사용하는 것과 다를 수 있습니다. 일부 API 메서드는 Node.js에서 작동하지만 브라우저 스크립트에서는 작동하지 않습니다. 마찬가지로 브라우저 스크립트에서는 작동하지만 Node.js

에서 작동하지 않는 API 메서드도 있습니다. 또한 일부 API를 성공적으로 사용할 수 있는지는 `File System (fs)` 모듈 등의 여러 Node.js 모듈을 가져와 사용하는 것과 같은 일반적인 Node.js 코딩 패턴에 익숙한지 여부에 달려 있습니다.

기본 제공 Node.js 모듈 사용

Node.js는 설치하지 않고도 사용할 수 있는 기본 제공 모듈 모음을 제공합니다. 이러한 모듈을 사용하려면 `require` 메서드로 객체를 생성하여 모듈 이름을 지정합니다. 예를 들어 기본 제공 HTTP 모듈을 포함시키려면 다음을 사용합니다.

```
var http = require('http');
```

모듈의 메서드가 해당 객체의 메서드인 것처럼 모듈의 메서드를 호출합니다. 예를 들어 다음은 HTML 파일을 읽는 코드입니다.

```
// include File System module
var fs = require('fs');
// Invoke readFile method
fs.readFile('index.html', function(err, data) {
  if (err) {
    throw err;
  } else {
    // Successful file read
  }
});
```

Node.js가 제공하는 모든 기본 제공 모듈의 전체 목록은 Node.js 웹 사이트의 Node.js v6.11.1 Documentation을 참조하세요.

NPM 패키지 사용

기본 제공 모듈 외에도 Node.js 패키지 관리자인 npm의 타사 코드를 포함 및 통합할 수 있습니다. 이는 오픈 소스 Node.js 패키지와 이러한 패키지를 설치하기 위한 명령줄 인터페이스의 리포지토리입니다. npm과 현재 사용 가능한 패키지 목록에 대한 자세한 내용은 https://www.npmjs.com을 참조하세요. GitHub에서 사용할 수 있는 추가 Node.js 패키지에 대해 알아볼 수도 있습니다.

AWS SDK for JavaScript에서 사용할 수 있는 npm 패키지의 한 예는 `browserify`입니다. 자세한 내용은 Browserify를 사용하여 SDK를 종속성으로 빌드을 참조하세요. 또 다른 예는 `webpack`입니다. 자세한 내용은 Webpack과 애플리케이션 번들링을 참조하세요.

주제

- Node.js에서 maxSockets 구성
- Node.js에서 연결 유지를 이용해 연결 재사용
- Node.js용 프록시 구성
- Node.js에서 인증서 번들 등록

Node.js에서 maxSockets 구성

Node.js에서 오리진당 최대 연결 수를 설정할 수 있습니다. `maxSockets`이 설정된 경우, 하위 HTTP 클라이언트가 요청을 대기열에 넣고 소켓이 사용 가능해지면 소켓에 요청을 할당합니다.

그러면 지정된 오리진에 한 번에 동시 요청할 수 있는 수의 상한을 설정할 수 있습니다. 이 값을 낮추면 수신하는 조절 또는 시간 초과 오류 수를 줄일 수 있습니다. 그러나 소켓이 사용 가능해질 때까지 요청이 대기되므로 메모리 사용량도 증가할 수 있습니다.

다음 예제에서는 생성하는 모든 서비스 객체에 대해 `maxSockets`를 설정하는 방법을 보여 줍니다. 이 예제에서는 각 서비스 엔드포인트에 최대 25개의 동시 연결을 허용합니다.

```
var AWS = require('aws-sdk');
var https = require('https');
var agent = new https.Agent({
   maxSockets: 25
});

AWS.config.update({
   httpOptions:{
      agent: agent
   }
});
```

서비스당 이와 동일한 작업을 수행할 수 있습니다.

```
var AWS = require('aws-sdk');
var https = require('https');
var agent = new https.Agent({
   maxSockets: 25
});
```

```
var dynamodb = new AWS.DynamoDB({
   apiVersion: '2012-08-10'
   httpOptions:{
      agent: agent
   }
});
```

기본값인 https를 사용하면 SDK가 maxSockets에서 globalAgent 값을 가져옵니다. maxSockets 값이 정의되지 않았거나 Infinity인 경우, SDK가 maxSockets 값을 50으로 가정합니다.

Node.js에서 maxSockets 설정에 대한 자세한 내용은 Node.js 온라인 설명서를 참조하세요.

Node.js에서 연결 유지를 이용해 연결 재사용

기본적으로 기본 Node.js HTTP/HTTPS 에이전트는 모든 새 요청에 대해 새로운 TCP 연결을 생성합니다. 새 연결 설정 비용이 발생하지 않게 하려면 기존 연결을 재사용하면 됩니다.

DynamoDB 쿼리와 같은 수명이 짧은 작업의 경우, TCP 연결 설정의 대기 시간 오버헤드가 작업 자체보다 클 수 있습니다. 또한 DynamoDB 유휴 시 암호화는 AWS KMS와 통합되므로 각 작업에 대해 새 AWS KMS 캐시 항목을 다시 설정해야 하는 데이터베이스에서 지연 시간이 발생할 수 있습니다.

TCP 연결을 재사용하도록 SDK for JavaScript를 구성하는 가장 쉬운 방법은 AWS_NODEJS_CONNECTION_REUSE_ENABLED 환경 변수를 1로 설정하는 것입니다. 이 기능은 2.463.0 릴리스에 추가되었습니다.

또는 다음 예와 같이 HTTP 또는 HTTPS 에이전트의 keepAlive 속성을 true로 설정할 수 있습니다.

```
const AWS = require('aws-sdk');
// http or https
const http = require('http');
const agent = new http.Agent({
  keepAlive: true,
// Infinity is read as 50 sockets
  maxSockets: Infinity
});

AWS.config.update({
  httpOptions: {
    agent
  }
```

```
});
```

다음 예에서는 DynamoDB 클라이언트에 대해서만 `keepAlive`를 설정하는 방법을 보여줍니다.

```
const AWS = require('aws-sdk')
// http or https
const https = require('https');
const agent = new https.Agent({
  keepAlive: true
});

const dynamodb = new AWS.DynamoDB({
  httpOptions: {
    agent
  }
});
```

`keepAlive`가 활성화된 경우 기본값이 1000ms인 `keepAliveMsecs`를 사용하여 TCP 연결 유지 패킷에 대한 초기 지연을 설정할 수도 있습니다. 자세한 내용은 Node.js 설명서를 참조하세요.

Node.js용 프록시 구성

인터넷에 직접 연결할 수 없는 경우, SDK for JavaScript가 proxy-agent와 같은 타사 HTTP 에이전트를 통해 HTTP 또는 HTTPS 프록시 사용을 지원합니다. proxy-agent를 설치하려면 명령줄에 다음을 입력합니다.

```
npm install proxy-agent --save
```

다른 프록시를 사용하기로 결정한 경우, 먼저 해당 프록시의 설치 및 구성 지침을 따릅니다. 애플리케이션에서 이 프록시 또는 다른 타사 프록시를 사용하려면 선택한 프록시를 지정하도록 `httpOptions`의 `AWS.Config` 속성을 설정해야 합니다. 이 예제에서는 `proxy-agent`를 보여줍니다.

```
var AWS = require("aws-sdk");
var ProxyAgent = require('proxy-agent').ProxyAgent;
AWS.config.update({
  httpOptions: { agent: new ProxyAgent('http://internal.proxy.com') }
});
```

다른 프록시 라이브러리에 대한 자세한 내용은 npm, the Node.js package manager를 참조하세요.

Node.js에서 인증서 번들 등록

Node.js용 기본 트러스트 스토어에는 AWS 서비스에 액세스하는 데 필요한 인증서가 포함되어 있습니다. 일부 경우에는 특정 인증서 집합만 포함하는 것이 좋을 수도 있습니다.

이 예제에서는 지정된 인증서가 제공되지 않은 경우 디스크의 특정 인증서를 사용하여 연결을 거부하는 `https.Agent`를 생성합니다. 그런 다음 새로 생성된 `https.Agent`를 사용하여 SDK 구성을 업데이트합니다.

```
var fs = require('fs');
var https = require('https');
var certs = [
  fs.readFileSync('/path/to/cert.pem')
];

AWS.config.update({
  httpOptions: {
    agent: new https.Agent({
      rejectUnauthorized: true,
      ca: certs
    })
  }
});
```

브라우저 스크립트 고려 사항

다음 주제에서는 브라우저 스크립트에서 AWS SDK for JavaScript를 사용할 때 특별히 고려해야 할 사항을 설명합니다.

주제

- 브라우저용 SDK 빌드
- CORS(Cross-Origin Resource Sharing)

브라우저용 SDK 빌드

SDK for JavaScript는 기본 서비스 집합에 대한 지원이 포함된 JavaScript 파일로 제공됩니다. 이 파일은 일반적으로 호스팅된 SDK 패키지를 참조하는 `<script>` 태그를 사용하여 브라우저 스크립트에 로드됩니다. 그러나 기본 집합 이외의 서비스에 대한 지원이 필요하거나 SDK를 사용자 지정해야 할 수 있습니다.

브라우저에서 CORS를 적용하는 환경 외부에서 SDK를 사용하며 SDK for JavaScript에서 제공하는 모든 서비스에 액세스하려는 경우 리포지토리를 복제하고 기본 호스팅 버전의 SDK를 빌드하는 동일한 빌드 도구를 실행하여 SDK의 사용자 지정 사본을 로컬에서 빌드할 수 있습니다. 다음 섹션에서는 추가 서비스 및 API 버전으로 SDK를 빌드하는 단계를 설명합니다.

주제

- SDK 빌더를 사용하여 SDK for JavaScript 빌드
- SDK 빌더를 사용하여 SDK for JavaScript 빌드
- 특정 서비스 및 API 버전 빌드
- Browserify를 사용하여 SDK를 종속성으로 빌드

SDK 빌더를 사용하여 SDK for JavaScript 빌드

AWS SDK for JavaScript의 자체 빌드를 생성하는 가장 쉬운 방법은 https://sdk.amazonaws.com/builder/js에서 SDK 빌더 웹 애플리케이션을 사용하는 것입니다. SDK 빌더를 사용하여 빌드에 포함시킬 서비스 및 서비스의 API 버전을 지정합니다.

서비스를 추가하거나 제거할 수 있는 시작 지점으로 Select all services(모든 서비스 선택)를 선택하거나 Select default services(기본 서비스 선택)를 선택합니다. Development(개발)를 선택하여 더 읽기 쉬운 코드를 만들거나 Minified(축소)를 선택하여 축소판 빌드를 만들어 배포합니다. 포함시킬 서비스와 버전을 선택한 후 Build(빌드)를 선택하여 사용자 지정 SDK를 빌드하고 다운로드합니다.

SDK 빌더를 사용하여 SDK for JavaScript 빌드

AWS CLI로 SDK for JavaScript를 빌드하려면 먼저 SDK 소스가 포함된 Git 리포지토리를 복제해야 합니다. 컴퓨터에 Git 및 Node.js가 설치되어 있어야 합니다.

먼저 GitHub에서 리포지토리를 복제하고 디렉터리를 다음 디렉터리로 변경합니다.

```
git clone https://github.com/aws/aws-sdk-js.git
cd aws-sdk-js
```

리포지토리를 복제한 후 SDK와 빌드 도구의 종속성 모듈을 다운로드합니다.

```
npm install
```

이제 SDK의 패키지 버전을 빌드할 수 있습니다.

명령줄에서 빌드

빌더 도구는 `dist-tools/browser-builder.js`에 있습니다. 다음을 입력하여 이 스크립트를 실행합니다.

```
node dist-tools/browser-builder.js > aws-sdk.js
```

이 명령은 aws-sdk.js 파일을 빌드합니다. 이 파일은 압축되어 있지 않습니다. 기본적으로 이 패키지에는 표준 서비스 집합만 포함되어 있습니다.

빌드 출력 축소

네트워크에서 데이터의 양을 줄이기 위해 축소라는 프로세스를 통해 JavaScript 파일을 압축할 수 있습니다. 축소는 사용자가 읽기 쉽도록 코드 실행에 영향을 주지 않는 설명, 불필요한 공백 및 기타 문자를 제거합니다. 빌더 도구로 압축되지 않았거나 축소된 출력을 생성할 수 있습니다. 빌드 출력을 축소하려면 `MINIFY` 환경 변수를 설정합니다.

```
MINIFY=1 node dist-tools/browser-builder.js > aws-sdk.js
```

특정 서비스 및 API 버전 빌드

SDK에 빌드할 서비스를 선택할 수 있습니다. 서비스를 선택하려면 쉼표로 구분되는 서비스 이름을 파라미터로 지정합니다. 예를 들어 Amazon S3 및 Amazon EC2만 빌드하려면 다음 명령을 사용합니다.

```
node dist-tools/browser-builder.js s3,ec2 > aws-sdk-s3-ec2.js
```

서비스 이름 뒤에 버전 이름을 추가하여 서비스 빌드의 특정 API 버전을 선택할 수도 있습니다. 예를 들어 Amazon DynamoDB의 API 버전을 모두 빌드하려면 다음 명령을 사용합니다.

```
node dist-tools/browser-builder.js dynamodb-2011-12-05,dynamodb-2012-08-10
```

서비스 식별자와 API 버전은 https://github.com/aws/aws-sdk-js/tree/master/apis의 서비스별 구성 파일에서 확인할 수 있습니다.

모든 서비스 빌드

`all` 파라미터를 포함시켜 모든 서비스와 API 버전을 빌드할 수 있습니다.

```
node dist-tools/browser-builder.js all > aws-sdk-full.js
```

특정 서비스 빌드

빌드에 포함된 선택한 서비스 집합을 사용자 지정하려면 원하는 서비스 목록이 포함된 Browserify 명령에 `AWS_SERVICES` 환경 변수를 전달합니다. 다음 예제로 Amazon EC2, Amazon S3 및 DynamoDB 서비스를 빌드합니다.

```
$ AWS_SERVICES=ec2,s3,dynamodb browserify index.js > browser-app.js
```

Browserify를 사용하여 SDK를 종속성으로 빌드

Node.js에는 타사 개발자의 코드와 기능을 포함시키기 위한 모듈 기반 메커니즘이 있습니다. 이 모듈식 접근 방식은 웹 브라우저에서 실행되는 JavaScript에서 기본 지원되지 않습니다. 그러나 Browserify라는 도구를 통해 Node.js 모듈 접근 방식을 사용하고 브라우저에서 Node.js용으로 작성된 모듈을 사용할 수 있습니다. Browserify는 브라우저 스크립트의 모듈 종속성을 브라우저에서 사용할 수 있는 단일의 독립형 JavaScript 파일에 빌드합니다.

Browserify를 사용하여 SDK를 모든 브라우저 스크립트의 라이브러리 종속성으로 빌드할 수 있습니다. 예를 들어 다음과 같은 Node.js 함수 코드에 SDK가 필요합니다.

```
var AWS = require('aws-sdk');
var s3 = new AWS.S3();
s3.listBuckets(function(err, data) { console.log(err, data); });
```

이 예제 코드는 Browserify를 사용하여 브라우저와 호환되는 버전으로 컴파일할 수 있습니다.

```
$ browserify index.js > browser-app.js
```

그러면 SDK 종속성을 포함하여 애플리케이션을 `browser-app.js`를 통해 브라우저에서 사용할 수 있게 됩니다.

Browserify에 대한 자세한 내용은 Browserify 웹 사이트를 참조하세요.

CORS(Cross-Origin Resource Sharing)

CORS(Cross-origin 리소스 공유)는 최신 웹 브라우저의 보안 기능입니다. 이 기능을 사용하면 웹 브라우저가 어떤 도메인이 외부 웹 사이트 또는 서비스를 요청할 수 있을지 협상할 수 있습니다. 대부분의 리소스 요청이 외부 도메인(예: 웹 서비스용 엔드포인트)으로 전송되기 때문에 AWS SDK for JavaScript를 사용해 브라우저 애플리케이션을 개발하는 경우 CORS는 중요한 고려 대상입니다. JavaScript 환경에서 CORS 보안을 적용하는 경우 이 서비스와 함께 CORS를 구성해야 합니다.

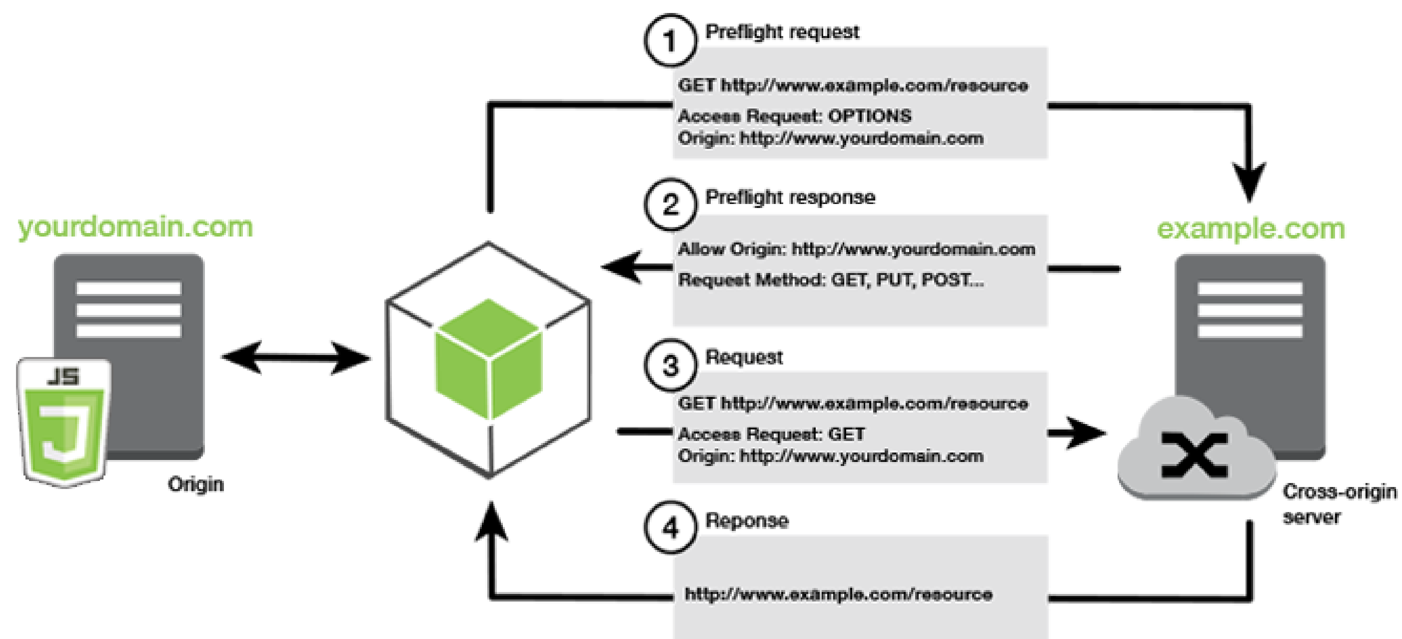
CORS는 다음을 기준으로 cross-origin 요청 시 리소스 공유 여부를 결정합니다.

- 요청을 수행한 특정 도메인
- 수행 중인 HTTP 요청 유형(GET, PUT, POST, DELETE 등)

CORS 작동 방식

가장 간단한 경우 브라우저 스크립트는 다른 도메인 내 서버의 리소스에 대해 GET 요청을 수행합니다. 요청이 GET 요청을 제출하도록 승인된 도메인에서 전송된 경우 해당 서버의 CORS 구성에 따라 cross-origin 서버는 요청된 리소스를 반환하여 응답합니다.

요청하는 도메인 또는 HTTP 요청 유형이 승인되지 않은 경우에는 요청이 거부됩니다. 그러나 CORS는 요청을 실제로 제출하기 전에 요청을 사전에 보낼 수 있습니다. 이 경우 preflight 요청은 `OPTIONS` 액세스 요청 작업을 전송하는 방식으로 이루어집니다. cross-origin 서버의 CORS 구성이 요청하는 도메인에 대한 액세스 권한을 부여하는 경우 서버에서는 요청하는 도메인이 요청한 리소스에 대해 수행 가능한 HTTP 요청 유형을 모두 나열하는 preflight 응답으로 회신합니다.



CORS 구성이 필요함

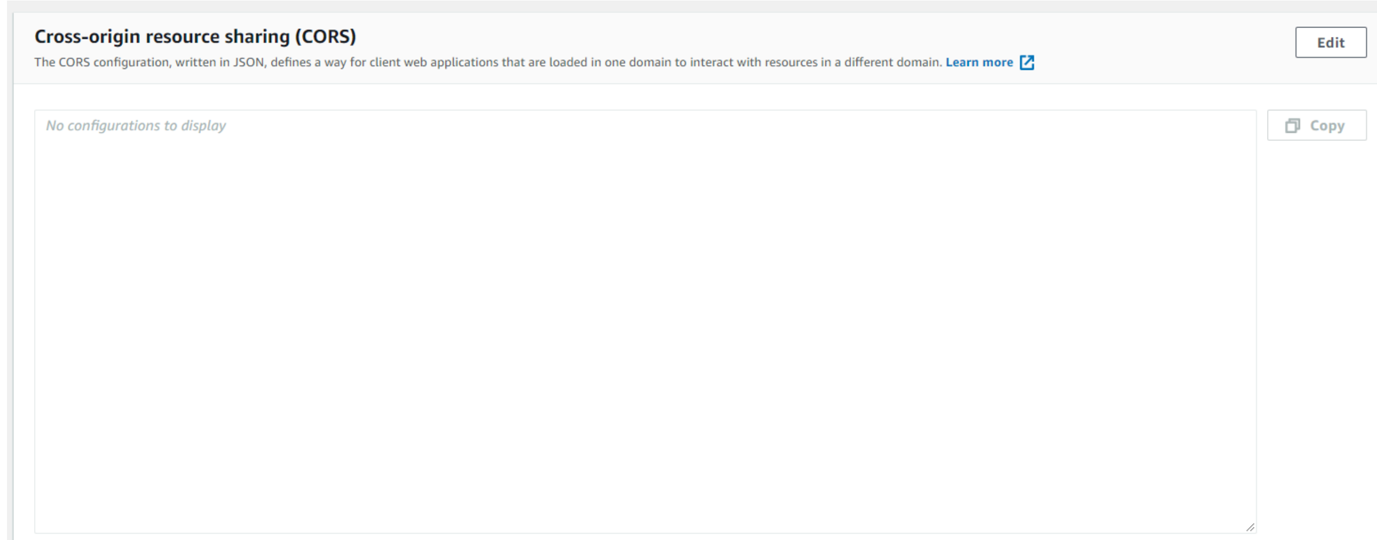
Amazon S3 버킷에서 작업을 수행하려면 먼저 해당 버킷에 CORS를 구성해야 합니다. 일부 JavaScript 환경에서는 CORS를 적용할 수 없기 때문에 CORS 구성이 불필요할 수 있습니다. 예를 들어 Amazon S3 버킷에서 애플리케이션을 호스팅하고 `*.s3.amazonaws.com` 또는 일부 다른 특정 엔드포인트에서 리소스에 액세스하는 경우에는 요청이 외부 도메인에 액세스하지 않습니다. 따라서 이

구성에는 CORS가 필요하지 않습니다. 이 경우에도 CORS는 Amazon S3 이외의 서비스에는 여전히 사용됩니다.

Amazon S3 버킷에 맞는 CORS 구성

Amazon S3 콘솔에서 CORS를 사용하도록 Amazon S3 버킷을 구성할 수 있습니다.

1. Amazon S3 콘솔에서 수정할 버킷을 선택합니다.
2. 권한 탭을 선택하고 아래로 스크롤하여 CORS(Cross-origin resource sharing) 패널로 이동합니다.



3. 수정을 선택하고 CORS 구성 편집기에 CORS 구성을 입력한 다음 저장을 선택합니다.

CORS 구성은 `<CORSRule>` 내에 일련의 규칙이 포함된 XML 파일입니다. 구성에는 규칙이 최대 100 개까지 있을 수 있습니다. 규칙은 다음 태그 중 하나로 정의합니다.

- `<AllowedOrigin>`: 교차 도메인 요청을 수행하도록 허용하는 도메인 오리진을 지정합니다.
- `<AllowedMethod>`: 교차 도메인 요청에서 허용하는 요청 유형(GET, PUT, POST, DELETE, HEAD)을 지정합니다.
- `<AllowedHeader>`: preflight 요청에서 허용되는 헤더를 지정합니다.

구성의 예는 Amazon Simple Storage Service 사용 설명서의 [버킷 CORS 구성 방법] 섹션을 참조하세요.

CORS 구성의 예

`<AllowedOrigin>`의 범위를 웹 사이트의 도메인으로 지정하는 것이 좋긴 하지만 다음 CORS 구성 샘플을 사용하면 사용자는 `example.org` 도메인에서 버킷 내 객체를 보거나, 추가 또는 제거하거나, 업데이트할 수 있습니다. 모든 도메인을 허용하려면 `"*"`를 지정할 수 있습니다.

Important

새 S3 콘솔에서 CORS 구성은 JSON이어야 합니다.

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<CORSConfiguration xmlns="http://s3.amazonaws.com/doc/2006-03-01/">
  <CORSRule>
    <AllowedOrigin>https://example.org</AllowedOrigin>
    <AllowedMethod>HEAD</AllowedMethod>
    <AllowedMethod>GET</AllowedMethod>
    <AllowedMethod>PUT</AllowedMethod>
    <AllowedMethod>POST</AllowedMethod>
    <AllowedMethod>DELETE</AllowedMethod>
    <AllowedHeader>*</AllowedHeader>
    <ExposeHeader>ETag</ExposeHeader>
    <ExposeHeader>x-amz-meta-custom-header</ExposeHeader>
  </CORSRule>
</CORSConfiguration>
```

JSON

```
[
    {
        "AllowedHeaders": [
            "*"
        ],
        "AllowedMethods": [
            "HEAD",
            "GET",
            "PUT",
            "POST",
            "DELETE"
```

```
        ],
        "AllowedOrigins": [
            "https://www.example.org"
        ],
        "ExposeHeaders": [
             "ETag",
             "x-amz-meta-custom-header"]
    }
]
```

이 구성은 사용자가 버킷에 대해 작업을 수행하도록 허용하지 않고, 브라우저의 보안 모델이 Amazon S3에 대한 요청을 허용하도록 합니다. 권한은 버킷 권한 또는 IAM 역할 권한을 통해 구성해야 합니다.

`ExposeHeader`를 사용하면 SDK가 Amazon S3에서 반환된 응답 헤더를 읽도록 할 수 있습니다. 예를 들어, PUT 또는 멀티파트 업로드에서 `ETag` 헤더를 읽으려면 이전 예제에서처럼 구성에 `ExposeHeader` 태그를 포함해야 합니다. SDK는 CORS 구성을 통해 노출된 헤더에만 액세스할 수 있습니다. 객체에 대해 메타데이터를 설정한 경우 값은 접두사 `x-amz-meta-`가 붙은 헤더(예: `x-amz-meta-my-custom-header`)로 반환되며 동일한 방식으로 노출되어야 합니다.

Webpack과 애플리케이션 번들링

코드 모듈에서 브라우저 스크립트 또는 Node.js로 작성된 웹 애플리케이션을 사용하면 종속성이 생성됩니다. 이러한 코드 모듈에는 자체 종속성이 있을 수 있어 애플리케이션 작동에 필요한 상호 연결된 모듈 모음이 생깁니다. 종속성을 관리하려면 webpack과 같은 모듈 번들러를 사용할 수 있습니다.

webpack 모듈 번들러는 애플리케이션 코드를 구문 분석하여 `import` 또는 `require` 명령문을 검색해 애플리케이션에 필요한 모든 자산이 포함된 번들을 생성합니다. 따라서 웹 페이지를 통해 자산을 쉽게 제공할 수 있습니다. SDK for JavaScript는 출력 번들에 포함할 종속성 중 하나로 webpack에 포함될 수 있습니다.

webpack에 대한 자세한 내용은 GitHub의 webpack 모듈 번들러를 참조하세요.

Webpack 설치

webpack 모듈 번들러를 설치하려면 먼저 Node.js 패키지 관리자인 npm이 설치되어 있어야 합니다. 다음 명령을 입력하여 webpack CLI 및 JavaScript 모듈을 설치합니다.

```
npm install webpack
```

또한 JSON 파일을 로드할 수 있도록 하려면 webpack 플러그인을 설치해야 할 수도 있습니다. 다음 명령을 입력하여 JSON 로더 플러그인을 설치합니다.

```
npm install json-loader
```

Webpack 구성

기본적으로 webpack은 프로젝트의 루트 디렉터리에서 JavaScript 파일인 `webpack.config.js`를 검색합니다. 이 파일은 구성 옵션을 지정합니다. 다음은 `webpack.config.js` 구성 파일의 예입니다.

```
// Import path for resolving file paths
var path = require('path');
module.exports = {
  // Specify the entry point for our app.
  entry: [
    path.join(__dirname, 'browser.js')
  ],
  // Specify the output file containing our bundled code
  output: {
    path: __dirname,
    filename: 'bundle.js'
  },
  module: {
    /**
      * Tell webpack how to load 'json' files.
      * When webpack encounters a 'require()' statement
      * where a 'json' file is being imported, it will use
      * the json-loader.
      */
    loaders: [
      {
        test: /\.json$/,
        loaders: ['json']
      }
    ]
  }
}
```

이 예제에서는 `browser.js`가 진입점으로 지정됩니다. 이 진입점은 webpack이 가져온 모듈 검색을 시작하기 위해 사용하는 파일입니다. 출력의 파일 이름은 `bundle.js`로 지정합니다. 출력 파일에는 애플리케이션에서 실행해야 하는 JavaScript가 모두 포함되어 있습니다. 진입점에 지정된 코드가 SDK

for JavaScript와 같은 다른 모듈을 가져오거나 필요로 하는 경우 해당 코드는 구성에서 이를 지정할 필요 없이 번들링됩니다.

앞서 설치된 `json-loader` 플러그인의 구성은 webpack에서 JSON 파일을 가져오는 방식을 지정합니다. 기본적으로 webpack은 JavaScript만 지원하지만 로더를 사용하여 다른 파일 유형을 가져오기 위한 지원을 추가합니다. SDK for JavaScript에서는 JSON 파일의 사용을 확장하기 때문에 `json-loader`가 포함되지 않은 경우 번들 생성 시 webpack에서 오류가 발생합니다.

Webpack 실행

webpack을 사용하기 위한 애플리케이션을 빌드하려면 `package.json` 파일의 `scripts` 객체에 다음을 추가합니다.

```
"build": "webpack"
```

다음은 webpack 추가를 보여주는 예제 `package.json`입니다.

```
{
  "name": "aws-webpack",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "webpack"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "aws-sdk": "^2.6.1"
  },
  "devDependencies": {
    "json-loader": "^0.5.4",
    "webpack": "^1.13.2"
  }
}
```

애플리케이션을 빌드하려면 다음 명령을 입력합니다.

```
npm run build
```

webpack 모듈 번들러가 프로젝트의 루트 디렉터리에 지정한 JavaScript 파일을 생성합니다.

Webpack 번들 사용

브라우저 스크립트에서 번들을 사용하기 위해 다음 예제에서처럼 `<script>` 태그를 사용해 번들을 포함할 수 있습니다.

```
<!DOCTYPE html>
<html>
    <head>
        <title>AWS SDK with webpack</title>
    </head>
    <body>
        <div id="list"></div>
        <script src="bundle.js"></script>
    </body>
</html>
```

개별 서비스 가져오기

webpack 사용의 이점 중 하나는 코드의 종속성을 구문 분석해 애플리케이션에 필요한 코드만 번들링한다는 점입니다. SDK for JavaScript를 사용하는 경우에는 애플리케이션에서 실제 사용하는 SDK의 부분만 번들링해 webpack 출력 크기를 크게 줄일 수 있습니다.

Amazon S3 서비스 객체를 생성하는 데 사용되는 다음 코드 예제를 사용하면 좋습니다.

```
// Import the AWS SDK
var AWS = require('aws-sdk');

// Set credentials and Region
// This can also be done directly on the service client
AWS.config.update({region: 'us-west-1', credentials: {YOUR_CREDENTIALS}});

var s3 = new AWS.S3({apiVersion: '2006-03-01'});
```

`require()` 함수는 전체 SDK를 지정합니다. 이 코드를 사용하여 생성된 webpack 번들은 전체 SDK를 포함할 수 있지만 Amazon S3 클라이언트 클래스만 사용되는 경우에는 전체 SDK가 필요하지 않습니다. Amazon S3 서비스에 필요한 SDK의 일부만 포함하면 번들의 크기를 상당히 줄일 수 있습니다. Amazon S3 서비스 객체에 대한 구성 데이터를 설정할 수 있기 때문에 구성 설정에 전체 SDK가 필요하지 않습니다.

SDK의 Amazon S3 부분만 포함한 경우 코드는 다음과 같습니다.

```
// Import the Amazon S3 service client
var S3 = require('aws-sdk/clients/s3');

// Set credentials and Region
var s3 = new S3({
    apiVersion: '2006-03-01',
    region: 'us-west-1',
    credentials: {YOUR_CREDENTIALS}
  });
```

Node.js에 대한 번들링

webpack을 사용하면 구성에서 Node.js를 대상으로 지정하여 Node.js에서 실행되는 번들을 생성할 수 있습니다.

```
target: "node"
```

이는 디스크 공간이 제한된 환경에서 Node.js 애플리케이션을 실행하는 경우 유용합니다. 다음은 출력 대상으로 Node.js가 지정된 `webpack.config.js` 구성의 예입니다.

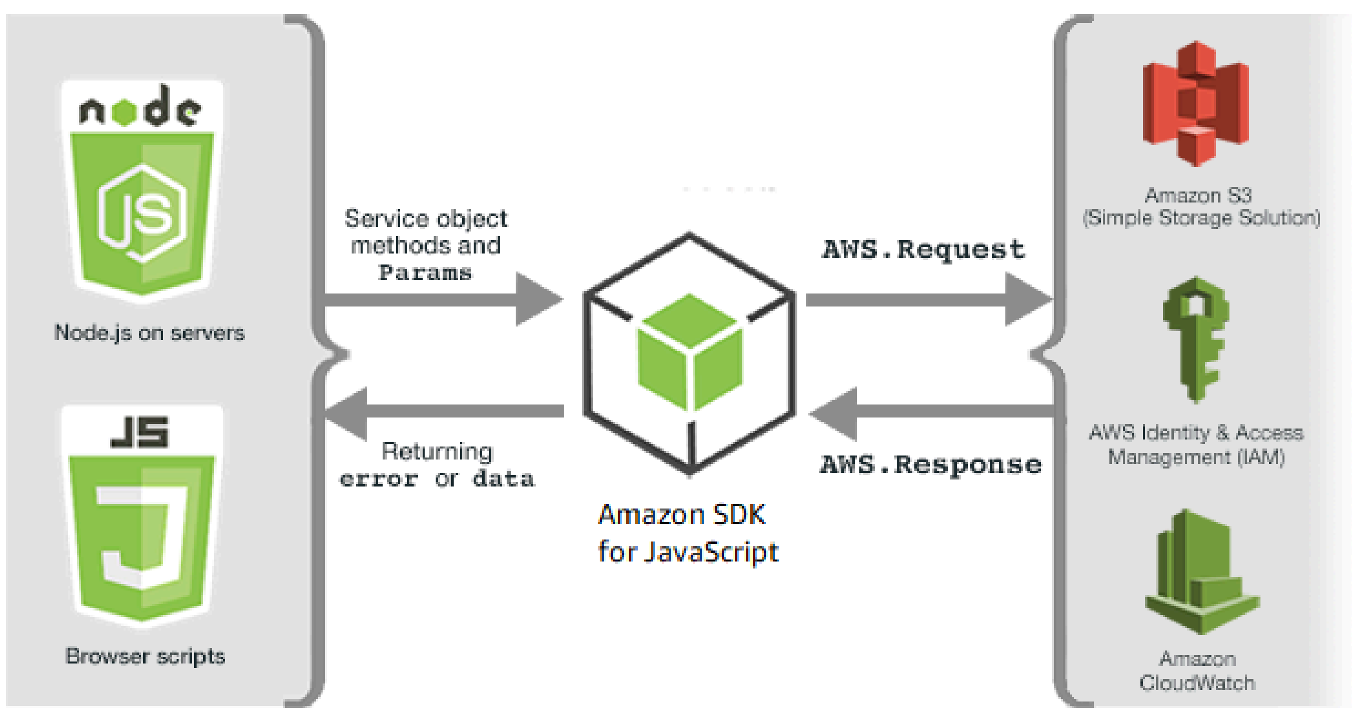
```
// Import path for resolving file paths
var path = require('path');
module.exports = {
  // Specify the entry point for our app
  entry: [
    path.join(__dirname, 'node.js')
  ],
  // Specify the output file containing our bundled code
  output: {
    path: __dirname,
    filename: 'bundle.js'
  },
  // Let webpack know to generate a Node.js bundle
  target: "node",
  module: {
    /**
      * Tell webpack how to load JSON files.
      * When webpack encounters a 'require()' statement
      * where a JSON file is being imported, it will use
      * the json-loader
```

```
      */
    loaders: [
      {
        test: /\.json$/,
        loaders: ['json']
      }
    ]
  }
}
```

SDK for JavaScript에서 서비스 사용

AWS SDK for JavaScript에서는 클라이언트 클래스 수집을 통해 지원하는 서비스에 액세스할 수 있습니다. 이러한 클라이언트 클래스에서는 일반적으로 서비스 객체라고 부르는 서비스 인터페이스 객체를 생성합니다. 지원되는 각 AWS 서비스에는 서비스 기능 및 리소스 사용을 위해 하위 수준 API를 제공하는 하나 이상의 클라이언트 클래스가 있습니다. 예를 들어 Amazon DynamoDB API는 `AWS.DynamoDB` 클래스를 통해 사용할 수 있습니다.

SDK for JavaScript를 통해 노출되는 서비스는 요청-응답 패턴에 따라 호출 애플리케이션과 메시지를 교환합니다. 이 패턴에서 서비스를 호출하는 코드는 해당 서비스의 엔트포인트에 HTTP/HTTPS 요청을 제출합니다. 이 요청에는 호출 중인 특정 기능을 성공적으로 호출하는 데 필요한 파라미터가 포함되어 있습니다. 호출된 서비스는 다시 요청자에게 보낼 응답을 생성합니다. 이 응답에는 작업에 성공에 성공한 경우에는 데이터가, 작업에 실패한 경우에는 오류 정보가 포함됩니다.



AWS 서비스 호출에는 모든 시도를 비롯하여 서비스 객체에 대한 작업의 전체 요청 및 응답 수명 주기가 포함되어 있습니다. 요청은 `AWS.Request` 객체가 SDK에서 캡슐화합니다. 응답은 `AWS.Response` 객체가 SDK에서 캡슐화하고, 호출 함수 또는 JavaScript promise 등과 같은 여러 기법 중 하나를 통해 요청자에게 제공됩니다.

주제

- 서비스 객체 생성 및 호출
- AWS SDK for JavaScript 호출 로깅
- 비동기식 서비스 호출
- 응답 객체 사용
- JSON 작업
- AWS SDK for JavaScript v2의 재시도 전략

서비스 객체 생성 및 호출

JavaScript API는 사용 가능한 AWS 서비스를 대부분 지원합니다. JavaScript API의 각 서비스 클래스는 서비스 내 모든 API 호출에 대한 액세스를 제공합니다. JavaScript API의 서비스 클래스, 작업 및 파라미터에 대한 자세한 내용은 API 참조를 참조하세요.

Node.js에서 SDK를 사용하는 경우에는 `require`를 사용하여 애플리케이션에 SDK 패키지를 추가합니다. 이 패키지는 현재 모든 서비스에 대한 지원을 제공합니다.

```
var AWS = require('aws-sdk');
```

브라우저 JavaScript와 함께 SDK를 사용하는 경우에는 AWS 호스팅 SDK 패키지를 사용하여 브라우저 스크립트에 SDK 패키지를 로드합니다. SDK 패키지를 로드하려면 다음 `<script>` 요소를 추가합니다.

```
<script src="https://sdk.amazonaws.com/js/aws-sdk-SDK_VERSION_NUMBER.min.js"></script>
```

AWS SDK for JavaScript API 참조 가이드의 SDK for JavaScript에 대한 API 참조 섹션에서 현재 SDK_VERSION_NUMBER를 찾을 수 있습니다.

호스팅된 기본 SDK 패키지는 사용 가능한 AWS 서비스의 하위 세트에 대한 지원을 제공합니다. 브라우저를 위해 호스팅되는 SDK 패키지의 기본 서비스 목록은 API 참조의 지원되는 서비스를 참조하세요. CORS 보안 확인이 비활성화된 경우에는 다른 서비스와 함께 SDK를 사용할 수 있습니다. 이러한 경우에는 필요한 추가 서비스를 포함하도록 사용자 지정 SDK 버전을 빌드할 수 있습니다. 사용자 지정 SDK 버전 빌드에 대한 자세한 내용은 브라우저용 SDK 빌드 섹션을 참조하세요.

개별 서비스 필요

이전에 설명한 것처럼 SDK for JavaScript가 필요하면 코드에 전체 SDK를 포함해야 합니다. 또는 코드에서 사용하는 개별 서비스만 필요하도록 선택할 수도 있습니다. Amazon S3 서비스 객체를 생성하려면 아래의 코드를 사용하는 것이 좋습니다.

```
// Import the AWS SDK
var AWS = require('aws-sdk');

// Set credentials and Region
// This can also be done directly on the service client
AWS.config.update({region: 'us-west-1', credentials: {YOUR_CREDENTIALS}});

var s3 = new AWS.S3({apiVersion: '2006-03-01'});
```

이전 예제에서 `require` 함수는 전체 SDK를 지정했습니다. Amazon S3 서비스에 필요한 SDK의 일부만 포함하면 네트워크를 통해 전송되는 코드의 양과 코드의 메모리 오버헤드가 크게 줄어들 수 있습니다. 개별 서비스를 필요로 하기 위해서는 표시된 것처럼 서비스 생성자를 포함해 `require` 함수를 모두 소문자로 호출합니다.

```
require('aws-sdk/clients/SERVICE');
```

SDK의 Amazon S3 부분만 포함했을 때의 이전 Amazon S3 서비스 객체를 생성하는 코드는 아래와 유사합니다.

```
// Import the Amazon S3 service client
var S3 = require('aws-sdk/clients/s3');

// Set credentials and Region
var s3 = new S3({
    apiVersion: '2006-03-01',
    region: 'us-west-1',
    credentials: {YOUR_CREDENTIALS}
  });
```

모든 서비스가 연결되지 않은 상태에서 전역 AWS 네임스페이스에 액세스할 수 있습니다.

```
require('aws-sdk/global');
```

이는 예를 들어 모든 서비스에 동일한 자격 증명을 제공하기 위해 여러 개별 서비스 간에 동일한 구성을 적용하는 경우 유용한 기법입니다. 개별 서비스만 필요하면 Node.js에서 로드 시간과 메모리 사용량이 줄어듭니다. Browserify 또는 webpack 등과 같은 번들링 도구를 사용해 수행하는 경우 개별 서비스만 필요로 하면 SDK가 전체 크기의 몇 분의 일 수준으로 줄어듭니다. 이는 IoT 디바이스 또는 Lambda 함수와 같이 메모리 또는 디스크 공간이 제한된 환경에서 유용합니다.

서비스 객체 생성

JavaScript API를 통해 서비스 기능에 액세스하려면 기본 클라이언트 클래스에서 제공하는 기능 세트에 액세스하기 위해 통하는 서비스 객체를 먼저 생성합니다. 일반적으로 각 서비스에는 클라이언트 클래스 하나가 제공되지만 일부 서비스는 여러 클라이언트 클래스 간에 기능에 대한 액세스를 나눕니다.

기능을 사용하려면 해당 기능에 대한 액세스를 제공하는 클래스의 인스턴스를 생성해야 합니다. 다음 예제에서는 `AWS.DynamoDB` 클라이언트 클래스에서 DynamoDB에 대한 서비스 객체 생성을 보여줍니다.

```
var dynamodb = new AWS.DynamoDB({apiVersion: '2012-08-10'});
```

기본적으로 서비스 객체는 SDK를 구성하는 데에도 사용된 전역 설정으로 구성됩니다. 그러나 해당 서비스 객체에 고유한 런타임 구성 데이터를 사용해 서비스 객체를 구성할 수 있습니다. 서비스별 구성 데이터는 전역 구성 설정을 적용한 다음에 적용됩니다.

다음 예제에서 Amazon EC2 서비스 객체는 특정 리전에 대한 구성 또는 전역 구성을 사용해 생성됩니다.

```
var ec2 = new AWS.EC2({region: 'us-west-2', apiVersion: '2014-10-01'});
```

개별 서비스 객체에 적용된 서비스별 구성을 지원하는 것 외에도 지정된 클래스의 새로 생성된 서비스 객체 모두에 서비스별 구성을 적용할 수도 있습니다. 예를 들어, 미국 서부(오레곤)(`us-west-2`) 리전을 사용하도록 클래스에서 생성된 모든 서비스 객체를 구성하려면 `AWS.config` 전역 구성 객체에 다음을 추가합니다.

```
AWS.config.ec2 = {region: 'us-west-2', apiVersion: '2016-04-01'};
```

서비스 객체의 API 버전 잠금

객체를 생성할 때 `apiVersion` 옵션을 지정하여 서비스의 특정 API 버전으로 서비스 객체를 잠글 수 있습니다. 다음 예제에서는 특정 API 버전으로 잠긴 DynamoDB 서비스 객체가 생성됩니다.

```
var dynamodb = new AWS.DynamoDB({apiVersion: '2011-12-05'});
```

서비스 객체의 API 버전 잠금에 대한 자세한 내용은 API 버전 잠금 섹션을 참조하세요.

서비스 객체 파라미터 지정

서비스 객체의 메서드를 호출할 때 API에서 요구하는 대로 파라미터를 JSON으로 전달합니다. 예를 들어 Amazon S3에서, 지정된 버킷 및 키에 대한 객체를 가져오려면 `getObject` 메서드에 다음 파라미터를 전달합니다. JSON 파라미터 전달에 대한 자세한 내용은 JSON 작업 섹션을 참조하세요.

```
s3.getObject({Bucket: 'bucketName', Key: 'keyName'});
```

Amazon S3 파라미터에 관한 자세한 내용은 API 참조의 `Class: AWS.S3` 섹션을 참조하세요.

또한 `params` 파라미터를 사용하여 서비스 객체를 생성할 때 개별 파라미터에 값을 바인딩할 수 있습니다. 서비스 객체의 `params` 파라미터 값은 서비스 객체가 정의한 파라미터 값을 하나 이상 지정하는 맵입니다. 다음 예제에서는 `amzn-s3-demo-bucket` 버킷으로 바인딩된 Amazon S3 서비스 객체의 `Bucket` 파라미터를 보여줍니다.

```
var s3bucket = new AWS.S3({params: {Bucket: 'amzn-s3-demo-bucket'}, apiVersion:
 '2006-03-01' });
```

서비스 객체를 버킷으로 바인딩함으로써 `s3bucket` 서비스 객체는 `amzn-s3-demo-bucket` 파라미터 값을 후속 작업에 대해 더 이상 지정할 필요가 없는 기본값으로 취급합니다. 파라미터 값을 적용할 수 없는 작업에 객체를 사용하는 경우에는 모든 바인딩 파라미터 값이 무시됩니다. 새 값을 지정하여 서비스 객체를 호출하면 바인딩된 파라미터를 재정의할 수 있습니다.

```
var s3bucket = new AWS.S3({ params: {Bucket: 'amzn-s3-demo-bucket'}, apiVersion:
 '2006-03-01' });
s3bucket.getObject({Key: 'keyName'});
// ...
s3bucket.getObject({Bucket: 'amzn-s3-demo-bucket3', Key: 'keyOtherName'});
```

각 메서드에 사용 가능한 파라미터에 대한 자세한 내용은 API 참조에서 확인할 수 있습니다.

AWS SDK for JavaScript 호출 로깅

AWS SDK for JavaScript는 기본 제공 로거가 함께 제공되므로 SDK for JavaScript로 API 직접 호출을 로깅할 수 있습니다.

콘솔에서 이 로거를 켜고 로그 항목을 인쇄하려면 코드에 다음 명령문을 추가하십시오.

```
AWS.config.logger = console;
```

다음은 로그 출력의 예입니다.

```
[AWS s3 200 0.185s 0 retries] createMultipartUpload({ Bucket: 'amzn-s3-demo-logging-bucket', Key: 'issues_1704' })
```

타사 로거 사용

또한 로그 파일 또는 서버에 쓰기 위한 `log()` 또는 `write()` 작업이 있는 경우에는 타사 로거를 사용할 수도 있습니다. SDK for JavaScript에서 사용자 지정 로거를 사용하려면 지침에 따라 설치 및 설정해야 합니다.

브라우저 스크립트 또는 Node.js에서 사용할 수 있는 로거 중 하나는 logplease입니다. Node.js에서는 로그 파일에 로그 항목을 쓰도록 logplease를 구성할 수 있습니다. logplease는 webpack과 함께 사용할 수도 있습니다.

타사 로거를 사용하는 경우에는 로거를 `AWS.Config.logger`에 할당하기 전에 모든 옵션을 설정합니다. 예를 들어, 다음은 외부 로그 파일을 지정하고, logplease에 대한 로그 수준을 설정합니다.

```
// Require AWS Node.js SDK
const AWS = require('aws-sdk')
// Require logplease
const logplease = require('logplease');
// Set external log file option
logplease.setLogfile('debug.log');
// Set log level
logplease.setLogLevel('DEBUG');
// Create logger
const logger = logplease.create('logger name');
// Assign logger to SDK
AWS.config.logger = logger;
```

logplease에 대한 자세한 내용은 GitHub에서 [logplease Simple JavaScript Logger](https://github.com/haadcode/logplease)를 참조하세요.

비동기식 서비스 호출

SDK를 통해 수행한 모든 요청은 비동기식입니다. 브라우저 스크립트를 작성할 때 이 점을 항상 주의해야 합니다. 웹 브라우저에서 실행 중인 JavaScript에는 일반적으로 실행 스레드가 하나 뿐입니다. AWS

서비스에 대한 비동기식 호출 이후에 브라우저 스크립트는 계속해서 실행되고 그 과정에서 브라우저 스크립트에서 반환하기 전에 비동기식 결과를 사용하는 코드를 실행하려고 시도할 수 있습니다.

AWS 서비스에 대한 비동기식 호출에는 이러한 호출에 대한 관리가 필요합니다. 그래야 코드가 데이터를 사용할 수 있기 전에 데이터를 사용하려고 시도하지 않습니다. 이 섹션의 주제에서는 비동기식 호출 관리의 필요성과 비동기식 호출 관리에 사용할 수 있는 다양한 기법에 대해 자세히 다룹니다.

주제

- 비동기식 호출 관리
- 익명 콜백 함수 사용
- 요청 객체 이벤트 리스너 사용
- 비동기/대기 사용
- JavaScript Promises 사용

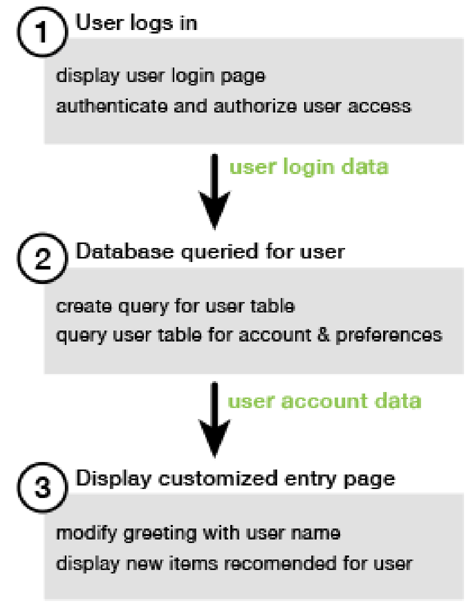
비동기식 호출 관리

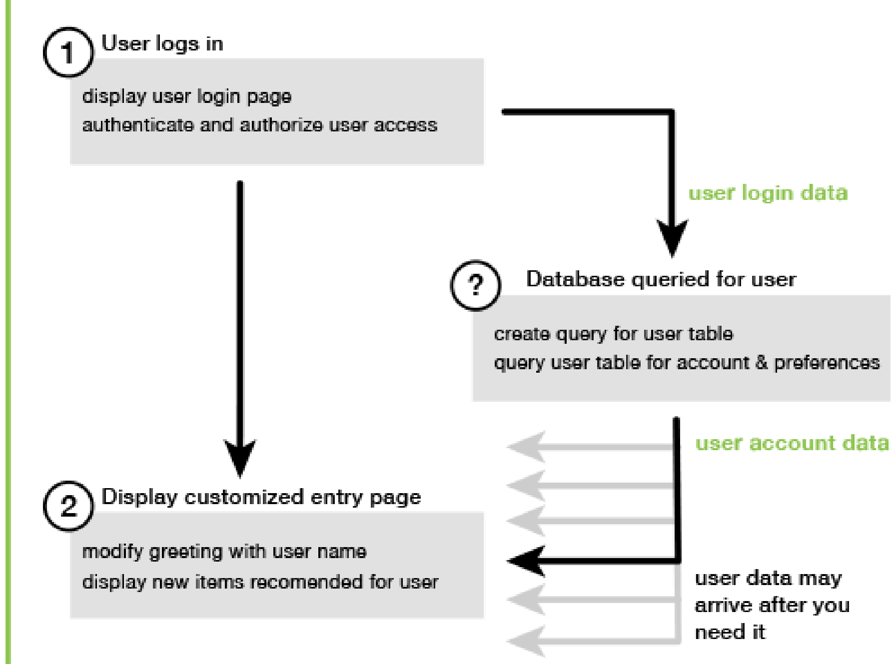
예를 들어, 전자 상거래 웹 사이트의 홈 페이지에서는 재방문 고객이 로그인할 수 있습니다. 로그인한 고객을 위한 혜택의 일부로 로그인 후 사이트에서는 고객의 특정 기본 설정에 맞춰 사이트를 맞춤화합니다. 사이트를 맞춤화하려면 다음을 수행해야 합니다.

1. 고객이 로그인하고 로그인 보안 인증으로 검증되어야 합니다.
2. 고객 데이터베이스에서 고객의 기본 설정이 요청됩니다.
3. 데이터베이스에서는 페이지 로드 전에 사이트를 맞춤화하는 데 사용되는 고객의 기본 설정을 제공합니다.

이러한 작업이 동기식으로 실행되면 다음 작업을 시작하기 전에 각 작업이 끝나야 합니다. 따라서 고객 기본 설정이 데이터베이스에서 반환될 때까지 웹 페이지 로딩을 완료할 수 없습니다. 그러나 데이터베이스 쿼리가 서버로 전송된 후 네트워크 병목 현상, 예외적으로 높은 데이터베이스 트래픽 또는 불안한 모바일 디바이스 연결 등으로 인해 고객 데이터 수신이 지연되거나 실패할 수 있습니다.

이러한 상황에서도 웹 사이트가 멈추지 않도록 하기 위해 데이터베이스를 비동기식을 호출합니다. 데이터베이스 호출을 실행한 후 비동기 요청을 보내면 코드가 계속해서 예상대로 실행됩니다. 비동기 호출의 응답을 적절하게 관리하지 못하면 데이터를 아직 사용할 수 없는데도 코드가 데이터베이스에서 다시 필요한 정보를 사용하려고 시도할 수 있습니다.





익명 콜백 함수 사용

AWS.Request 객체를 생성하는 각 서비스 객체 메서드는 익명 콜백 함수를 마지막 파라미터로 수락할 수 있습니다. 이 콜백 함수의 서명은 다음과 같습니다.

```
function(error, data) {
    // callback handling code
}
```

이 콜백 함수는 성공적인 응답 또는 오류 데이터 반환 시 실행됩니다. 메서드 호출에 성공하면 data 파라미터에서 응답 내용을 콜백 함수에 사용할 수 있습니다. 호출에 실패하면 error 파라미터에 자세한 실패 정보가 제공됩니다.

일반적으로 콜백 함수 내 코드는 오류가 있는지 테스트하는데, 오류가 반환되면 처리합니다. 오류가 반환되지 않으면 코드는 응답의 data 파라미터에서 데이터를 검색합니다. 콜백 함수의 기본 형식은 다음 예제와 같습니다.

```
function(error, data) {
    if (error) {
        // error handling code
        console.log(error);
```

```
    } else {
        // data handling code
        console.log(data);
    }
}
```

이전 예제에서는 오류 또는 반환되는 데이터에 대한 자세한 내용이 콘솔에 로깅됩니다. 다음은 서비스 객체에 대한 메서드 호출의 일부로 전달되는 콜백 함수를 보여주는 예제입니다.

```
new AWS.EC2({apiVersion: '2014-10-01'}).describeInstances(function(error, data) {
  if (error) {
    console.log(error); // an error occurred
  } else {
    console.log(data); // request succeeded
  }
});
```

요청 및 응답 객체에 액세스

콜백 함수 내에서 JavaScript 키워드 `this`는 대부분 서비스의 경우 기본 `AWS.Response` 객체를 가리킵니다. 다음 예제에서는 디버깅을 지원하기 위해 `httpResponse` 객체의 `AWS.Response` 속성이 콜백 함수 내에서 사용되어 원시 응답 데이터 및 헤더를 로깅합니다.

```
new AWS.EC2({apiVersion: '2014-10-01'}).describeInstances(function(error, data) {
  if (error) {
    console.log(error); // an error occurred
    // Using this keyword to access AWS.Response object and properties
    console.log("Response data and headers: " + JSON.stringify(this.httpResponse));
  } else {
    console.log(data); // request succeeded
  }
});
```

또한 `AWS.Response` 객체에는 원래 메서드 호출에서 보낸 `Request`가 포함된 `AWS.Request` 속성이 있으므로 수행된 요청에 대한 세부 정보에 액세스할 수도 있습니다.

요청 객체 이벤트 리스너 사용

서비스 객체 메서드를 호출할 때 익명 콜백 함수를 파라미터로 생성해 전달하지 않으면 해당 메서드 호출은 `AWS.Request` 메서드를 사용해 수동으로 전송해야 하는 `send` 객체를 생성합니다.

응답을 처리하려면 AWS.Request 객체에 대한 이벤트 리스너를 생성해 메서드 호출에 대한 콜백 함수를 등록해야 합니다. 다음 예제에서는 성공적인 반환을 위해 서비스 객체 메서드 및 이벤트 리스너를 호출하기 위한 AWS.Request 객체를 생성하는 방법을 보여줍니다.

```
// create the AWS.Request object
var request = new AWS.EC2({apiVersion: '2014-10-01'}).describeInstances();

// register a callback event handler
request.on('success', function(response) {
  // log the successful data response
  console.log(response.data);
});

// send the request
request.send();
```

send 객체에 대한 AWS.Request 메서드를 호출한 후에는 서비스 객체가 AWS.Response 객체를 수신할 때 이벤트 핸들러가 실행됩니다.

AWS.Request 객체에 대한 자세한 내용은 API 참조의 Class: AWS.Request 섹션을 참조하세요. AWS.Response 객체에 대한 자세한 내용은 응답 객체 사용 또는 API 참조의 Class: AWS.Response 섹션을 참조하세요.

여러 콜백 묶기

모든 요청 객체에 대해 여러 콜백을 등록할 수 있습니다. 다른 이벤트 또는 동일한 이벤트에 대해 여러 콜백을 등록할 수 있습니다. 또한 다음 예제에서처럼 콜백을 묶을 수 있습니다.

```
request.
  on('success', function(response) {
    console.log("Success!");
  }).
  on('error', function(response) {
    console.log("Error!");
  }).
  on('complete', function() {
    console.log("Always!");
  }).
  send();
```

객체 완료 이벤트 요청

`AWS.Request` 객체는 각 서비스 작업 메서드의 응답을 바탕으로 다음 완료 이벤트를 발생시킵니다.

- `success`
- `error`
- `complete`

이러한 이벤트에 대한 응답으로 콜백 함수를 등록할 수 있습니다. 모든 요청 객체 이벤트의 전체 목록을 보려면 API 참조의 [`Class: AWS.Request`] 클래스를 참조하세요.

성공 이벤트

`success` 이벤트는 서비스 객체에서 성공적인 응답을 수신하면 발생합니다. 다음은 성공 이벤트에 대한 콜백 함수를 등록하는 방법입니다.

```
request.on('success', function(response) {
  // event handler code
});
```

응답은 서비스의 직렬화된 응답 데이터가 포함된 `data` 속성을 제공합니다. 예를 들어, Amazon S3 서비스 객체의 `listBuckets` 메서드에 대한 직접접 호출은 다음과 같습니다.

```
s3.listBuckets.on('success', function(response) {
  console.log(response.data);
}).send();
```

이 호출은 응답을 반환한 다음 콘솔에 다음 `data`속성 내용을 출력합니다.

```
{ Owner: { ID: '...', DisplayName: '...' },
  Buckets:
   [ { Name: 'someBucketName', CreationDate: someCreationDate },
     { Name: 'otherBucketName', CreationDate: otherCreationDate } ],
  RequestId: '...' }
```

오류 이벤트

`error` 이벤트는 서비스 객체에서 오류 응답을 수신하면 발생합니다. 다음은 성공 이벤트에 대한 콜백 함수를 등록하는 방법입니다.

```
request.on('error', function(error, response) {
  // event handling code
});
```

error 이벤트가 발생하면 응답의 data 속성 값이 null이고 error 속성에는 오류 데이터가 포함됩니다. 연결된 error 객체는 등록된 콜백 함수에 첫 번째 파라미터로 전달됩니다. 예를 들어, 다음 코드에서는

```
s3.config.credentials.accessKeyId = 'invalid';
s3.listBuckets().on('error', function(error, response) {
  console.log(error);
}).send();
```

오류를 반환한 다음 콘솔에 다음 오류 데이터를 출력합니다.

```
{ code: 'Forbidden', message: null }
```

완료 이벤트

호출 결과 성공 또는 오류인지 여부에 상관 없이 complete 이벤트는 서비스 객체 호출이 완료되면 발생합니다. 다음은 성공 이벤트에 대한 콜백 함수를 등록하는 방법입니다.

```
request.on('complete', function(response) {
  // event handler code
});
```

complete 이벤트 콜백을 사용하면 성공 또는 오류 여부에 상관 없이 실행해야 하는 모든 요청 정리를 처리할 수 있습니다. complete 이벤트에 대한 콜백 내에서 응답 데이터를 사용하면 다음 예제에서 보여주는 것처럼 response.data 또는 response.error 속성 중 하나에 액세스하기 전에 먼저 두 속성을 확인합니다.

```
request.on('complete', function(response) {
  if (response.error) {
    // an error occurred, handle it
  } else {
    // we can use response.data here
  }
}).send();
```

객체 HTTP 이벤트 요청

AWS.Request 객체는 각 서비스 작업 메서드의 응답을 바탕으로 다음 HTTP 이벤트를 발생시킵니다.

- httpHeaders
- httpData
- httpUploadProgress
- httpDownloadProgress
- httpError
- httpDone

이러한 이벤트에 대한 응답으로 콜백 함수를 등록할 수 있습니다. 모든 요청 객체 이벤트의 전체 목록을 보려면 API 참조의 Class: AWS.Request 클래스를 참조하세요.

httpHeaders 이벤트

httpHeaders 이벤트는 원격 서버에서 헤더를 보낸 경우 발생합니다. 다음은 성공 이벤트에 대한 콜백 함수를 등록하는 방법입니다.

```
request.on('httpHeaders', function(statusCode, headers, response) {
  // event handling code
});
```

콜백 함수에 대한 statusCode 파라미터가 HTTP 상태 코드입니다. headers 파라미터에는 응답 헤더가 포함되어 있습니다.

httpData 이벤트

httpData 이벤트는 서버에서 응답 데이터 패킷을 스트리밍하기 위해 발생됩니다. 다음은 성공 이벤트에 대한 콜백 함수를 등록하는 방법입니다.

```
request.on('httpData', function(chunk, response) {
  // event handling code
});
```

이 이벤트는 일반적으로 전체 응답을 메모리로 로드하는 것이 적절하지 않은 경우 대용량 응답을 청크로 수신하는 데 사용됩니다. 이 이벤트에는 서버의 실제 데이터 중 일부가 포함되어 있는 추가 chunk 파라미터가 있습니다.

httpData 이벤트에 대한 콜백을 등록한 경우 응답의 data 속성에는 요청에 대해 직렬화된 전체 출력이 포함됩니다. 기본 제공 핸들러에 대한 추가 구문 분석 및 메모리 오버헤드가 없는 경우에는 기본 httpData 리스너를 제거해야 합니다.

httpUploadProgress 및 httpDownloadProgress 이벤트

httpUploadProgress 이벤트는 HTTP 응답이 추가 데이터를 업로드한 경우 발생합니다. 마찬가지로, httpDownloadProgress 이벤트는 HTTP 요청에 추가 데이터를 다운로드한 경우 발생합니다. 다음은 이러한 이벤트에 대한 콜백 함수를 등록하는 방법입니다.

```
request.on('httpUploadProgress', function(progress, response) {
  // event handling code
})
.on('httpDownloadProgress', function(progress, response) {
  // event handling code
});
```

콜백 함수에 대한 progress 파라미터에는 로드된 총 요청 바이트와 함께 객체가 포함되어 있습니다.

httpError 이벤트

httpError 이벤트는 HTTP 요청에 실패한 경우 발생합니다. 다음은 성공 이벤트에 대한 콜백 함수를 등록하는 방법입니다.

```
request.on('httpError', function(error, response) {
  // event handling code
});
```

콜백 함수에 대한 error 파라미터에는 발생한 오류가 포함되어 있습니다.

httpDone 이벤트

httpDone 이벤트는 서버가 데이터 전송을 마치면 발생합니다. 다음은 성공 이벤트에 대한 콜백 함수를 등록하는 방법입니다.

```
request.on('httpDone', function(response) {
  // event handling code
});
```

비동기/대기 사용

AWS SDK for JavaScript를 직접 호출할 때 `async/await` 패턴을 사용할 수 있습니다. 콜백을 받는 대부분의 함수는 promise를 반환하지 않습니다. promise를 반환하는 `await` 함수만 사용하므로 `async/await` 패턴을 사용하려면 `.promise()` 메서드를 호출이 끝날 때까지 연결하고 콜백을 제거해야 합니다.

다음 예에서는 async/await를 사용하여 `us-west-2`의 모든 Amazon DynamoDB 테이블을 나열합니다.

```
var AWS = require("aws-sdk");
//Create an Amazon DynamoDB client service object.
dbClient = new AWS.DynamoDB({ region: "us-west-2" });
// Call DynamoDB to list existing tables
const run = async () => {
  try {
    const results = await dbClient.listTables({}).promise();
    console.log(results.TableNames.join("\n"));
  } catch (err) {
    console.error(err);
  }
};
run();
```

> **Note**
>
> 모든 브라우저가 async/await를 지원하는 것은 아닙니다. async/await를 지원하는 브라우저 목록은 Async functions를 참조하세요.

JavaScript Promises 사용

`AWS.Request.promise` 메서드는 서비스 작업을 호출하고 콜백을 사용하지 않고 비동기식 흐름을 관리하는 방법을 제공합니다. Node.js 및 브라우저 스크립트에서 `AWS.Request` 객체는 콜백 함수 없이 서비스 작업이 호출된 경우 반환됩니다. 요청의 `send` 메서드를 호출하여 서비스를 호출할 수 있습니다.

그러나 `AWS.Request.promise`는 서비스 호출을 즉시 시작하고 응답 `data` 속성을 사용하여 이행되었거나 응답 `error` 속성을 사용하여 거부된 promise를 반환합니다.

```
var request = new AWS.EC2({apiVersion: '2014-10-01'}).describeInstances();

// create the promise object
var promise = request.promise();

// handle promise's fulfilled/rejected states
promise.then(
  function(data) {
    /* process the data */
  },
  function(error) {
    /* handle the error */
  }
);
```

다음 예제에서는 `data` 객체로 이행되었거나 `error` 객체로 거부된 promise를 반환합니다. promise를 사용하는 경우 단일 콜백이 오류 탐지를 담당하지 않습니다. 대신 요청 성공 또는 실패를 바탕으로 올바른 콜백이 호출됩니다.

```
var s3 = new AWS.S3({apiVersion: '2006-03-01', region: 'us-west-2'});
var params = {
  Bucket: 'bucket',
  Key: 'example2.txt',
  Body: 'Uploaded text using the promise-based method!'
};
var putObjectPromise = s3.putObject(params).promise();
putObjectPromise.then(function(data) {
  console.log('Success');
}).catch(function(err) {
  console.log(err);
});
```

여러 Promise 조정

경우에 따라 코드는 여러 비동기식 호출이 모두 성공적으로 반환된 경우에만 조치가 필요한 여러 비동기식 호출을 수행해야 합니다. promise를 사용하지 않고 개별 비동기 메서드 호출을 관리하는 경우 `all` 메서드를 사용하는 추가 promise를 생성할 수 있습니다. 이 메서드는 메서드에 전달한 promise 배열이 이행되는 경우 umbrella promise를 이행합니다. 콜백 함수는 `all` 메서드에 전달되는 promises의 값 배열로 전달됩니다.

다음 예에서 AWS Lambda 함수는 Amazon DynamoDB에 대한 비동기 직접 호출을 3개 수행해야 하는데 각 직접 호출에 대한 promise를 이행한 후에만 완료할 수 있습니다.

```
Promise.all([firstPromise, secondPromise, thirdPromise]).then(function(values) {

  console.log("Value 0 is " + values[0].toString);
  console.log("Value 1 is " + values[1].toString);
  console.log("Value 2 is " + values[2].toString);

  // return the result to the caller of the Lambda function
  callback(null, values);
});
```

Promise에 대한 브라우저 및 Node.js 지원

기본 JavaScript promise(ECMAScript 2015)에 대한 지원은 모드가 실행되는 JavaScript 엔진 및 버전에 따라 달라집니다. 코드를 실행해야 하는 각 환경에서 JavaScript promise에 대한 지원을 확인하려면 GitHub에서 ECMAScript 호환성 표를 참조하세요.

기타 Promise 구현 사용

ECMAScript 2015의 기본 promise 구현 이외에 다음을 포함해 타사 promise 라이브러리도 사용할 수 있습니다.

- bluebird
- RSVP
- Q.

이러한 선택적 promise 라이브러리는 ECMAScript 5 및 ECMAScript 2015에서 기본 promise 구현을 지원하지 않는 환경에서 코드를 실행해야 하는 경우 유용할 수 있습니다.

타사 promise 라이브러리를 사용하려면 전역 구성 객체의 `setPromisesDependency` 메서드를 호출하여 SDK에 대한 promise 종속성을 설정해야 합니다. 브라우저 스크립트에서 SDK를 로드하기 전에 타사 promise 라이브러리를 로드해야 합니다. 다음 예제에서는 bluebird promise 라이브러리에서 구현을 사용하도록 SDK가 구성되었습니다.

```
AWS.config.setPromisesDependency(require('bluebird'));
```

JavaScript 엔진의 기본 promise 구현을 사용하기 위해 반환하려면 `setPromisesDependency`를 다시 호출하여 라이브러리 이름 대신 `null`을 전달합니다.

응답 객체 사용

서비스 객체 메서드가 호출되면 해당 메서드는 콜백 함수에 자신을 전달해 `AWS.Response` 객체를 반환합니다. `AWS.Response` 객체의 속성을 통해 응답 내용에 액세스합니다. 응답 내용에 액세스하는 데 사용할 수 있는 `AWS.Response` 객체의 속성은 다음과 같이 두 가지가 있습니다.

- `data` 속성
- `error` 속성

표준 콜백 메커니즘을 사용하는 경우 이러한 두 가지 속성은 다음 예에서처럼 익명 콜백 함수에 대한 파라미터로 제공됩니다.

```
function(error, data) {
    if (error) {
        // error handling code
        console.log(error);
    } else {
        // data handling code
        console.log(data);
    }
}
```

응답 객체에서 반환된 데이터에 액세스

`data` 객체의 `AWS.Response` 속성에는 서비스 요청에서 반환되는 직렬화된 데이터가 포함되어 있습니다. 요청에 성공하면 `data` 속성에는 반환되는 데이터에 대한 맵을 포함하는 객체가 들어 있습니다. 오류가 발생하면 `data` 속성이 null일 수 있습니다.

다음은 게임의 일부로 사용할 이미지 파일의 이름을 검색하기 위해 DynamoDB 테이블의 `getItem` 메서드를 호출하는 예제입니다.

```
// Initialize parameters needed to call DynamoDB
var slotParams = {
    Key : {'slotPosition' : {N: '0'}},
    TableName : 'slotWheels',
```

```
    ProjectionExpression: 'imageFile'
};

// prepare request object for call to DynamoDB
var request = new AWS.DynamoDB({region: 'us-west-2', apiVersion:
 '2012-08-10'}).getItem(slotParams);
// log the name of the image file to load in the slot machine
request.on('success', function(response) {
    // logs a value like "cherries.jpg" returned from DynamoDB
    console.log(response.data.Item.imageFile.S);
});
// submit DynamoDB request
request.send();
```

이 예제의 경우 DynamoDB 테이블은 슬롯 머신을 당긴 결과를 `slotParams`의 파라미터가 지정한 대로 보여주는 이미지 조회 테이블입니다.

`getItem` 메서드 호출에 성공하면 `AWS.Response` 객체의 `data` 속성에는 DynamoDB에서 반환한 `Item` 객체가 포함되어 있습니다. 반환된 데이터는 요청의 `ProjectionExpression` 파라미터에 따라 액세스되는데, 이번 경우에서 이 파라미터는 `imageFile` 객체의 `Item` 멤버입니다. `imageFile` 멤버가 문자열 값을 가지고 있기 때문에 `S`의 `imageFile` 하위 멤버 값을 통해 이미지의 파일 이름에 액세스합니다.

반환된 데이터를 통해 페이징

경우에 따라 서비스 요청에서 반환하는 `data` 속성의 내용이 여러 페이지에 걸쳐 있을 수 있습니다. `response.nextPage` 메서드를 호출하여 데이터의 다음 페이지에 액세스할 수 있습니다. 이 메서드는 새 요청을 보냅니다. 이 요청에 대한 응답은 콜백 또는 성공 및 오류 리스너를 사용해 캡처할 수 있습니다.

`response.hasNextPage` 메서드를 호출하여 서비스 요청에서 반환한 데이터에 추가 페이지가 있는지 확인할 수 있습니다. 이 메서드는 부울을 반환하여 `response.nextPage` 호출 시 추가 데이터를 반환하는지 여부를 나타냅니다.

```
s3.listObjects({Bucket: 'bucket'}).on('success', function handlePage(response) {
    // do something with response.data
    if (response.hasNextPage()) {
        response.nextPage().on('success', handlePage).send();
    }
}).send();
```

응답 객체에서 오류 정보에 액세스

error 객체의 AWS.Response 속성에는 서비스 오류 또는 전송 오류 이벤트 발생 시 확인 가능한 오류 데이터가 포함되어 있습니다. 오류는 다음과 같은 양식으로 반환됩니다.

```
{ code: 'SHORT_UNIQUE_ERROR_CODE', message: 'a descriptive error message' }
```

오류 발생 시 data 속성의 값은 null입니다. 실패 상태일 수 있는 이벤트를 처리하는 경우에는 error 속성 값에 액세스하려고 시도하기 전에 data 속성을 설정했는지 항상 확인하십시오.

발신 요청 객체에 액세스

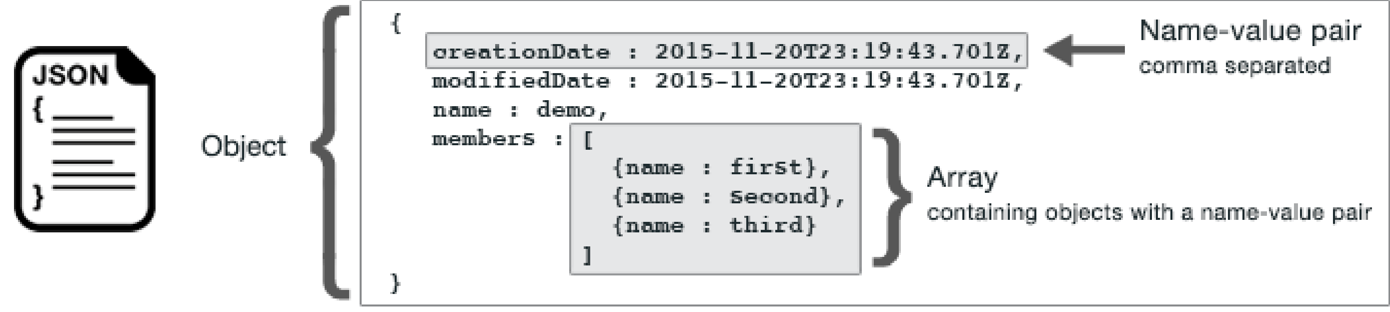
request 속성은 발신 AWS.Request 객체에 대한 액세스를 제공합니다. 이는 원래 AWS.Request 객체가 보낸 원래 파라미터에 액세스하기 위해 해당 객체를 참조하는 경우 유용합니다. 다음 예제에서 request 속성은 원래 서비스 요청의 Key 파라미터에 액세스하는 데 사용됩니다.

```
s3.getObject({Bucket: 'bucket', Key: 'key'}).on('success', function(response) {
   console.log("Key was", response.request.params.Key);
}).send();
```

JSON 작업

JSON은 인간 및 머신 둘 다 판독 가능한 데이터를 교환하기 위한 형식입니다. JSON이라는 이름이 JavaScript Object Notation의 약어이긴 하지만 JSON의 형식은 프로그래밍 언어와 관련이 없습니다.

SDK for JavaScript에서는 JSON을 사용하여 요청 시 서비스 객체에 데이터를 전송하고 서비스 객체에서 데이터를 JSON으로 수신합니다. JSON에 대한 자세한 내용은 json.org를 참조하세요.



JSON은 다음 두 가지 방식으로 데이터를 나타냅니다.

- 객체: 순서가 지정되지 안은 이름-값 쌍 모음. 객체는 여는 중괄호({)와 닫는 중괄호(}) 내에서 정의됩니다. 각 이름-값 쌍은 이름으로 시작하고 뒤에 콜론과 값이 옵니다. 이름-값 페어는 쉼표로 구분됩니다.
- 배열: 순서가 지정된 값 모음. 배열은 여는 대괄호([)와 닫는 대괄호(]) 안에 정의됩니다. 배열의 항목들은 쉼표로 구분됩니다.

다음은 객체가 카드 게임의 카드로 표현되는 객체 배열이 포함된 JSON 객체의 예입니다. 각 카드는 두 개의 이름-값 페어로 정의되는데, 하나는 하드를 식별하기 위한 고유한 값을 지정하고, 다른 하나는 해당하는 카드 이미지를 가리키는 URL을 지정합니다.

```
var cards = [{"CardID":"defaultname", "Image":"defaulturl"},
  {"CardID":"defaultname", "Image":"defaulturl"},
  {"CardID":"defaultname", "Image":"defaulturl"},
  {"CardID":"defaultname", "Image":"defaulturl"},
  {"CardID":"defaultname", "Image":"defaulturl"}];
```

서비스 객체 파라미터로서 JSON

다음은 Lambda 서비스 객체에 대한 호출의 파라미터를 정의하는 데 사용되는 간단한 JSON의 예입니다.

```
var pullParams = {
   FunctionName : 'slotPull',
   InvocationType : 'RequestResponse',
   LogType : 'None'
};
```

`pullParams` 객체는 여는 중괄호와 닫는 중괄호 내에서 쉼표로 구분된 이름-값 페어 3개로 정의됩니다. 서비스 객체 메서드에 파라미터를 제공하는 경우 이름은 호출하려는 서비스 객체 메서드에 대한 파라미터 이름으로 결정됩니다. Lambda 함수를 간접 호출할 때 `FunctionName`, `InvocationType`, `LogType`은 Lambda 서비스 객체에서 `invoke` 메서드를 직접 호출하는 데 사용되는 파라미터입니다.

서비스 객체 메서드 호출에 파라미터를 전달할 때 Lambda 함수를 호출하는 다음 예와 같이 메서드 호출에 JSON 객체를 제공합니다.

```
lambda = new AWS.Lambda({region: 'us-west-2', apiVersion: '2015-03-31'});
// create JSON object for service call parameters
var pullParams = {
```

```
    FunctionName : 'slotPull',
    InvocationType : 'RequestResponse',
    LogType : 'None'
};
// invoke Lambda function, passing JSON object
lambda.invoke(pullParams, function(err, data) {
    if (err) {
        console.log(err);
    } else {
        console.log(data);
    }
});
```

JSON으로 데이터 반환

JSON에서는 동시에 여러 값을 전송해야 하는 애플리케이션의 부분 간에 데이터를 전달하는 표준 방식을 제공합니다. API 내 클라이언트 클래스의 메서드는 일반적으로 콜백 함수로 전달되는 `data` 파라미터에 JSON을 반환합니다. 예를 들어, 다음은 Amazon S3 클라이언트 클래스의 `getBucketCors` 메서드에 대한 호출입니다.

```
// call S3 to retrieve CORS configuration for selected bucket
s3.getBucketCors(bucketParams, function(err, data) {
  if (err) {
    console.log(err);
  } else if (data) {
    console.log(JSON.stringify(data));
  }
});
```

`data`의 값은 JSON 객체이며 이 예에서는 지정된 Amazon S3 버킷에 대한 현재 CORS 구성을 설명하는 JSON입니다.

```
{
   "CORSRules": [
      {
          "AllowedHeaders":["*"],
          "AllowedMethods":["POST","GET","PUT","DELETE","HEAD"],
          "AllowedOrigins":["*"],
          "ExposeHeaders":[],
          "MaxAgeSeconds":3000
      }
   ]
```

```
}
```

AWS SDK for JavaScript v2의 재시도 전략

DNS 서버, 스위치, 로드 밸런서 등 수많은 네트워크 구성 요소들은 요청이 이루어지는 모든 단계에서 오류를 일으킬 수 있습니다. 네트워크 환경에서는 클라이언트 애플리케이션의 재시도 기술이 이러한 오류 응답을 처리하는 데 가장 많이 사용되고 있습니다. 이 기술은 애플리케이션의 신뢰성을 높일 뿐만 아니라 개발자의 운영 비용을 절감합니다. AWS SDK는 AWS 요청에 대한 자동 재시도 로직을 구현합니다.

지수 백오프 기반 재시도 동작

AWS SDK for JavaScript v2는 흐름 제어를 개선하기 위해 전체 지터(jitter)와 함께 지수 백오프를 사용하여 재시도 로직을 구현합니다. 지수 백오프의 기본 개념은 오류 응답이 연이어 나올 때마다 재시도 간 대기 시간을 점진적으로 늘리는 것입니다. 지터(무작위 지연)는 연속 충돌을 방지하는 데 사용됩니다.

v2에서 재시도 지연 테스트

v2에서 재시도 지연을 테스트하기 위해 node_modules/aws-sdk/lib/event_listeners.js의 코드가 다음과 같이 가변 지연에 있는 `console.log` 값으로 업데이트되었습니다.

```
// delay < 0 is a signal from customBackoff to skip retries
if (willRetry && delay >= 0) {
  resp.error = null;
  console.log('retry delay: ' + delay);
  setTimeout(done, delay);
} else {
  done();
}
```

기본 구성으로 지연 재시도

AWS SDK 클라이언트에서 모든 작업에 대한 지연을 테스트할 수 있습니다. 다음 코드를 사용하여 DynamoDB 클라이언트에서 `listTables` 작업을 호출합니다.

```
import AWS from "aws-sdk";

const region = "us-east-1";
```

```
const client = new AWS.DynamoDB({ region });
await client.listTables({}).promise();
```

재시도를 테스트하기 위해 테스트 코드를 실행하는 디바이스에서 인터넷 연결을 해제하여 `NetworkingError`를 시뮬레이션합니다. 또한 사용자 지정 오류를 반환하도록 프록시를 설정할 수 있습니다.

코드를 실행할 때 다음과 같이 지터와 함께 지수 백오프를 사용하여 재시도 지연을 확인할 수 있습니다.

```
retry delay: 7.39361151766359
retry delay: 9.0672860785882
retry delay: 134.89340825668168
retry delay: 398.53559817403965
retry delay: 523.8076165896343
retry delay: 1323.8789643058465
```

재시도는 지터를 사용하므로 예제 코드를 실행할 때 다른 값을 얻을 수 있습니다.

사용자 지정 기본으로 지연 재시도

AWS SDK for JavaScript v2를 사용하면 작업 재시도에 대한 지수 백오프에 사용할 사용자 지정 기본 밀리초를 전달할 수 있습니다. DynamoDB를 제외한 모든 서비스의 기본값은 100ms이며, 여기서 기본값은 50ms입니다.

다음과 같이 사용자 지정 기본값이 1,000ms인 재시도를 테스트합니다.

```
...
const client = new AWS.DynamoDB({ region, retryDelayOptions: { base: 1000 } });
...
```

테스트 코드를 실행하는 디바이스에서 인터넷 연결을 해제하여 `NetworkingError`를 시뮬레이션합니다. 기본값이 50ms 또는 100ms인 이전 실행에 비해 재시도 지연 값이 더 높다는 것을 알 수 있습니다.

```
retry delay: 356.2841549924913
retry delay: 1183.5216495444615
retry delay: 2266.997988094194
retry delay: 1244.6948354966453
retry delay: 4200.323030066383
```

재시도는 지터를 사용하므로 예제 코드를 실행할 때 다른 값을 얻을 수 있습니다.

사용자 지정 백오프 알고리즘을 사용하여 지연 재시도

또한 AWS SDK for JavaScript v2를 사용하면 재시도 횟수와 오류를 수락하고 지연 시간을 밀리초 단위로 반환하는 사용자 지정 백오프 함수를 전달할 수 있습니다. 결과가 0이 아닌 음수 값이면 더 이상 재시도하지 않습니다.

다음과 같이 기본값이 200ms인 선형 백오프를 사용하는 사용자 지정 백오프 함수를 테스트합니다.

```
...
const client = new AWS.DynamoDB({
  region,
  retryDelayOptions: { customBackoff: (count, error) => (count + 1) * 200 },
});
...
```

테스트 코드를 실행하는 디바이스에서 인터넷 연결을 해제하여 `NetworkingError`를 시뮬레이션합니다. 재시도 지연 값은 200의 배수입니다.

```
retry delay: 200
retry delay: 400
retry delay: 600
retry delay: 800
retry delay: 1000
```

SDK for JavaScript 코드 예제

이 단원의 주제에는 다양한 서비스의 API와 함께 AWS SDK for JavaScript를 사용하여 일반적인 작업을 수행하는 방법을 보여 주는 예제가 포함됩니다.

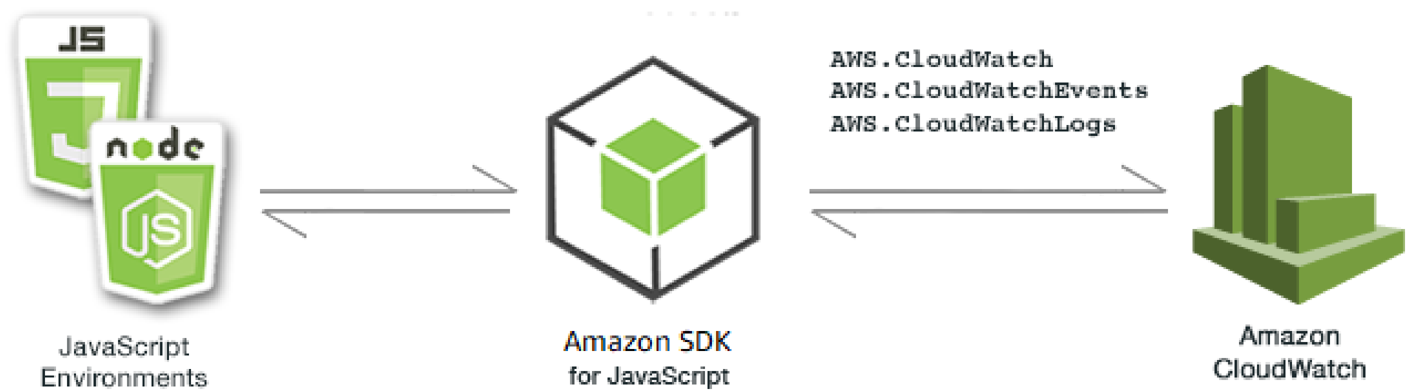
GitHub의 AWS 설명서 코드 예제 리포지토리에서 이러한 예제의 소스 코드와 다른 코드를 찾습니다. AWS 설명서 팀이 생성을 고려할 수 있는 새 코드 예제를 제안하려면 새 요청을 생성합니다. 이 팀은 개별 API 호출만 다루는 간단한 코드 조각에 비해 광범위한 시나리오 및 사용 사례를 다루는 코드를 생성하려고 합니다. 지침은 배포 가이드라인의 코드 작성 섹션을 참조하세요.

주제

- Amazon CloudWatch 예제
- Amazon DynamoDB 예제
- Amazon EC2 예제
- AWS Elemental MediaConvert 예제
- AWS IAM 예제
- Amazon Kinesis 예제
- Amazon S3 예제
- Amazon Simple Email Services 예제
- Amazon Simple Notification Service 예
- Amazon SQS 예제

Amazon CloudWatch 예제

Amazon CloudWatch(CloudWatch)는 Amazon Web Services 리소스 및 AWS에서 실행되는 애플리케이션을 실시간으로 모니터링하는 웹 서비스입니다. CloudWatch를 사용하여 리소스 및 애플리케이션에 대해 측정할 수 있는 변수인 지표를 수집하고 추적할 수 있습니다. CloudWatch 경보는 알림을 보내거나 정의한 규칙을 기준으로 모니터링하는 리소스를 자동으로 변경합니다.



CloudWatch용 JavaScript API는 `AWS.CloudWatch`, `AWS.CloudWatchEvents`, `AWS.CloudWatchLogs` 클라이언트 클래스를 통해 노출됩니다. CloudWatch 클라이언트 클래스 사용에 대한 자세한 내용은 API 참조의 `Class: AWS.CloudWatch`, `Class: AWS.CloudWatchEvents`, `Class: AWS.CloudWatchLogs` 섹션을 참조하세요.

주제

- Amazon CloudWatch 경보 생성
- Amazon CloudWatch 경보 작업 사용
- Amazon CloudWatch에서 지표 가져오기
- Amazon CloudWatch Events에 이벤트 전송
- Amazon CloudWatch Logs 구독 필터 사용

Amazon CloudWatch 경보 생성

이 Node.js 코드 예제는 다음을 보여 줍니다.

- CloudWatch 경보에 대한 기본 정보를 가져오는 방법
- CloudWatch 경보를 생성하고 삭제하는 방법

시나리오

경보는 지정한 기간에 단일 지표를 감시하고 여러 기간에 지정된 임곗값에 대한 지표 값을 기준으로 작업을 하나 이상 수행합니다.

이 예제에서는 일련의 Node.js 모듈을 사용하여 CloudWatch에서 경보를 생성합니다. Node.js 모듈은 SDK for JavaScript로 `AWS.CloudWatch` 클라이언트 클래스의 다음 메서드를 사용하여 알람을 생성합니다.

- `describeAlarms`
- `putMetricAlarm`
- `deleteAlarms`

CloudWatch 경보에 대한 자세한 내용은 Amazon CloudWatch 사용 설명서의 Amazon CloudWatch 경보 생성을 참조하세요.

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 이러한 작업들을 완료해야 합니다.

- Node.js를 설치합니다. Node.js 설치에 대한 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 공유 자격 증명 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.

경보 설명

파일 이름이 `cw_describealarms.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. CloudWatch에 액세스하려면 `AWS.CloudWatch` 서비스 객체를 생성합니다. 경보 설명을 가져오기 위한 파라미터를 담을 JSON 객체를 생성하여 `INSUFFICIENT_DATA` 상태의 설명으로 반환되는 경보를 제한합니다. 그런 다음 `describeAlarms` 서비스 객체의 `AWS.CloudWatch` 메서드를 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create CloudWatch service object
```

```
var cw = new AWS.CloudWatch({ apiVersion: "2010-08-01" });

cw.describeAlarms({ StateValue: "INSUFFICIENT_DATA" }, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    // List the names of all current alarms in the console
    data.MetricAlarms.forEach(function (item, index, array) {
      console.log(item.AlarmName);
    });
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node cw_describealarms.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

CloudWatch 지표에 대한 경보 생성

파일 이름이 `cw_putmetricalarm.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. CloudWatch에 액세스하려면 `AWS.CloudWatch` 서비스 객체를 생성합니다. 지표(이 경우 Amazon EC2 인스턴스의 CPU 사용률)를 기반으로 경보를 생성하는 데 필요한 파라미터를 담을 JSON 객체를 생성합니다. 나머지 파라미터는 지표가 임계값인 70퍼센트를 초과할 때 경보가 트리거되도록 설정됩니다. 그런 다음 `describeAlarms` 서비스 객체의 `AWS.CloudWatch` 메서드를 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create CloudWatch service object
var cw = new AWS.CloudWatch({ apiVersion: "2010-08-01" });

var params = {
  AlarmName: "Web_Server_CPU_Utilization",
  ComparisonOperator: "GreaterThanThreshold",
  EvaluationPeriods: 1,
  MetricName: "CPUUtilization",
```

```
  Namespace: "AWS/EC2",
  Period: 60,
  Statistic: "Average",
  Threshold: 70.0,
  ActionsEnabled: false,
  AlarmDescription: "Alarm when server CPU exceeds 70%",
  Dimensions: [
    {
      Name: "InstanceId",
      Value: "INSTANCE_ID",
    },
  ],
  Unit: "Percent",
};

cw.putMetricAlarm(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node cw_putmetricalarm.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

경보 삭제

파일 이름이 `cw_deletealarms.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. CloudWatch에 액세스하려면 `AWS.CloudWatch` 서비스 객체를 생성합니다. 삭제할 경보의 이름을 담을 JSON 객체를 생성합니다. 그런 다음 `deleteAlarms` 서비스 객체의 `AWS.CloudWatch` 메서드를 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create CloudWatch service object
```

```
var cw = new AWS.CloudWatch({ apiVersion: "2010-08-01" });

var params = {
  AlarmNames: ["Web_Server_CPU_Utilization"],
};

cw.deleteAlarms(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node cw_deletealarms.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

Amazon CloudWatch 경보 작업 사용

이 Node.js 코드 예제는 다음을 보여 줍니다.

- CloudWatch 경보를 기반으로 Amazon EC2 인스턴스의 상태를 자동으로 변경하는 방법

시나리오

경보 작업을 사용하면 Amazon EC2 인스턴스를 자동으로 중지, 종료, 재부팅 또는 복구하는 경보를 만들 수 있습니다. 인스턴스를 더는 실행할 필요가 없을 때 중지 또는 종료 작업을 사용할 수 있습니다. 재부팅 및 복구 작업을 사용하여 인스턴스를 자동으로 재부팅할 수 있습니다.

이 예제에서는 Amazon EC2 인스턴스의 재부팅을 트리거하는 CloudWatch의 경보 작업을 정의하는 데 일련의 Node.js 모듈이 사용됩니다. Node.js 모듈은 SDK for JavaScript로 `CloudWatch` 클라이언트 클래스의 다음 메서드를 사용하여 Amazon EC2 인스턴스를 관리합니다.

- enableAlarmActions
- disableAlarmActions

CloudWatch 경보 작업에 대한 자세한 내용을 알아보려면 Amazon CloudWatch 사용 설명서의 인스턴스를 중지, 종료, 재부팅 또는 복구하는 경보 생성을 참조하세요.

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 이러한 작업들을 완료해야 합니다.

- Node.js를 설치합니다. Node.js 설치에 대한 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 공유 자격 증명 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.
- Amazon EC2 인스턴스를 설명, 재부팅, 중지 또는 종료할 권한을 부여하는 정책이 있는 IAM 역할을 생성합니다. IAM 역할 생성에 관한 자세한 내용은 IAM 사용 설명서의 AWS 서비스에 대한 권한을 위임할 역할 생성 섹션을 참조하세요.

IAM 역할을 생성할 때 다음 역할 정책을 사용합니다.

JSON

```
{
   "Version":"2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "cloudwatch:Describe*",
            "ec2:Describe*",
            "ec2:RebootInstances",
            "ec2:StopInstances*",
            "ec2:TerminateInstances"
         ],
         "Resource": [
            "*"
         ]
      }
   ]
}
```

글로벌 구성 객체를 생성한 후 코드에 대한 리전을 설정하여 SDK for JavaScript를 구성합니다. 이 예제에서 리전이 `us-west-2`로 설정되어 있습니다.

```
// Load the SDK for JavaScript
var AWS = require('aws-sdk');
// Set the Region
AWS.config.update({region: 'us-west-2'});
```

경보 작업 생성 및 활성화

파일 이름이 `cw_enablealarmactions.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. CloudWatch에 액세스하려면 `AWS.CloudWatch` 서비스 객체를 생성합니다.

경보를 생성하기 위한 파라미터를 담을 JSON 객체를 생성하여 `ActionsEnabled`를 `true`로 지정하고 경보가 트리거할 작업에 대한 ARN 배열을 지정합니다. 경보가 없으면 생성하거나 경보가 있으면 업데이트하는 `putMetricAlarm` 서비스 객체의 `AWS.CloudWatch` 메서드를 호출합니다.

`putMetricAlarm`에 대한 콜백 함수가 성공적으로 완료되면 CloudWatch 경보의 이름이 포함된 JSON 객체를 생성합니다. `enableAlarmActions` 메서드를 호출하여 경보 작업을 활성화합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create CloudWatch service object
var cw = new AWS.CloudWatch({ apiVersion: "2010-08-01" });

var params = {
  AlarmName: "Web_Server_CPU_Utilization",
  ComparisonOperator: "GreaterThanThreshold",
  EvaluationPeriods: 1,
  MetricName: "CPUUtilization",
  Namespace: "AWS/EC2",
  Period: 60,
  Statistic: "Average",
  Threshold: 70.0,
  ActionsEnabled: true,
  AlarmActions: ["ACTION_ARN"],
  AlarmDescription: "Alarm when server CPU exceeds 70%",
  Dimensions: [
    {
```

```
      Name: "InstanceId",
      Value: "INSTANCE_ID",
    },
  ],
  Unit: "Percent",
};

cw.putMetricAlarm(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Alarm action added", data);
    var paramsEnableAlarmAction = {
      AlarmNames: [params.AlarmName],
    };
    cw.enableAlarmActions(paramsEnableAlarmAction, function (err, data) {
      if (err) {
        console.log("Error", err);
      } else {
        console.log("Alarm action enabled", data);
      }
    });
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node cw_enablealarmactions.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

경보 작업 비활성화

파일 이름이 `cw_disablealarmactions.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. CloudWatch에 액세스하려면 `AWS.CloudWatch` 서비스 객체를 생성합니다. CloudWatch 경보의 이름을 포함하는 JSON 객체를 생성합니다. `disableAlarmActions` 메서드를 호출하여 이 경보에 대한 작업을 비활성화합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });
```

```
// Create CloudWatch service object
var cw = new AWS.CloudWatch({ apiVersion: "2010-08-01" });

cw.disableAlarmActions(
  { AlarmNames: ["Web_Server_CPU_Utilization"] },
  function (err, data) {
    if (err) {
      console.log("Error", err);
    } else {
      console.log("Success", data);
    }
  }
);
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node cw_disablealarmactions.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

Amazon CloudWatch에서 지표 가져오기

이 Node.js 코드 예제는 다음을 보여 줍니다.

- 게시된 CloudWatch 지표의 목록을 검색하는 방법
- CloudWatch 지표에 데이터 포인트를 게시하는 방법

시나리오

지표는 시스템 성능에 대한 데이터입니다. Amazon EC2 인스턴스 같은 일부 리소스나 자체 애플리케이션 지표에 대한 세부 모니터링을 활성화할 수 있습니다.

이 예제에서는 일련의 Node.js 모듈을 사용하여 CloudWatch에서 지표를 가져오고 이벤트를 Amazon CloudWatch Events로 보냅니다. Node.js 모듈은 SDK for JavaScript로 `CloudWatch` 클라이언트 클래스의 다음 메서드를 사용하여 CloudWatch에서 지표를 가져옵니다.

- `listMetrics`
- `putMetricData`

CloudWatch 지표에 대한 자세한 내용은 Amazon CloudWatch 사용 설명서의 Amazon CloudWatch 지표 사용을 참조하세요.

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 이러한 작업들을 완료해야 합니다.

- Node.js를 설치합니다. Node.js 설치에 대한 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 공유 자격 증명 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.

지표 나열

파일 이름이 `cw_listmetrics.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. CloudWatch에 액세스하려면 `AWS.CloudWatch` 서비스 객체를 생성합니다. `AWS/Logs` 네임스페이스 내에 지표를 나열하는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. `listMetrics` 메서드를 호출하여 `IncomingLogEvents` 지표를 나열합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create CloudWatch service object
var cw = new AWS.CloudWatch({ apiVersion: "2010-08-01" });

var params = {
  Dimensions: [
    {
      Name: "LogGroupName" /* required */,
    },
  ],
  MetricName: "IncomingLogEvents",
  Namespace: "AWS/Logs",
};

cw.listMetrics(params, function (err, data) {
```

```
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Metrics", JSON.stringify(data.Metrics));
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node cw_listmetrics.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

사용자 지정 지표 제출

파일 이름이 `cw_putmetricdata.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. CloudWatch에 액세스하려면 `AWS.CloudWatch` 서비스 객체를 생성합니다. `PAGES_VISITED` 사용자 지정 지표에 대한 데이터 포인트를 제출하는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. `putMetricData` 메서드를 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create CloudWatch service object
var cw = new AWS.CloudWatch({ apiVersion: "2010-08-01" });

// Create parameters JSON for putMetricData
var params = {
  MetricData: [
    {
      MetricName: "PAGES_VISITED",
      Dimensions: [
        {
          Name: "UNIQUE_PAGES",
          Value: "URLS",
        },
      ],
      Unit: "None",
      Value: 1.0,
    },
```

```
  ],
  Namespace: "SITE/TRAFFIC",
};

cw.putMetricData(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", JSON.stringify(data));
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node cw_putmetricdata.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

Amazon CloudWatch Events에 이벤트 전송

이 Node.js 코드 예제는 다음을 보여 줍니다.

- 이벤트를 트리거하는 데 사용되는 규칙을 생성하고 업데이트하는 방법.
- 이벤트에 응답할 하나 이상의 대상을 정의하는 방법.
- 처리를 위해 대상과 일치하는 이벤트를 전송하는 방법.

시나리오

CloudWatch Events는 다양한 대상으로의 Amazon Web Services 리소스 변경 사항을 설명하는 시스템 이벤트의 스트림을 거의 실시간으로 전달합니다. 단순한 규칙을 사용하여 이벤트를 하나 이상의 대상 함수 또는 스트림에 일치시키고 라우팅할 수 있습니다.

이 예제에서는 일련의 Node.js 모듈을 사용하여 이벤트를 CloudWatch Events에 전송합니다. Node.js 모듈은 SDK for JavaScript로 `CloudWatchEvents` 클라이언트 클래스의 다음 메서드를 사용하여 인스턴스를 관리합니다.

- putRule
- putTargets
- putEvents

CloudWatch Events에 대한 자세한 내용은 Amazon CloudWatch Events 사용 설명서의 PutEvents로 이벤트 추가를 참조하세요.

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 이러한 작업들을 완료해야 합니다.

- Node.js를 설치합니다. Node.js 설치에 대한 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 공유 자격 증명 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.
- hello-world 블루프린트를 사용하는 Lambda 함수를 생성하여 이벤트의 대상 역할을 하도록 합니다. 방법을 알아보려면 Amazon CloudWatch Events 사용 설명서의 1단계: AWS Lambda 함수 생성을 참조하세요.
- CloudWatch Events에 권한을 부여하는 정책이 있고 신뢰할 수 있는 엔터티인 `events.amazonaws.com`을 포함하는 IAM 역할을 생성합니다. IAM 역할 생성에 관한 자세한 내용은 IAM 사용 설명서의 AWS 서비스에 대한 권한을 위임할 역할 생성 섹션을 참조하세요.

IAM 역할을 생성할 때 다음 역할 정책을 사용합니다.

JSON

```
{
   "Version":"2012-10-17",
   "Statement": [
      {
         "Sid": "CloudWatchEventsFullAccess",
         "Effect": "Allow",
         "Action": "events:*",
         "Resource": "*"
      },
      {
         "Sid": "IAMPassRoleForCloudWatchEvents",
         "Effect": "Allow",
```

```
            "Action": "iam:PassRole",
            "Resource": "arn:aws:iam::*:role/AWS_Events_Invoke_Targets"
         }
      ]
   }
```

IAM 역할을 생성할 때 다음과 같은 신뢰 관계를 사용합니다.

JSON

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

예약된 규칙 생성

파일 이름이 `cwe_putrule.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. CloudWatch Events에 액세스하려면 `AWS.CloudWatchEvents` 서비스 객체를 생성합니다. 새로운 예약 규칙을 지정하는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. 이 객체에는 다음이 포함됩니다.

- 규칙의 이름입니다
- 이전에 생성한 IAM 역할의 ARN
- 5분마다 규칙 트리거링을 예약할 표현식

`putRule` 메서드를 호출하여 규칙을 생성합니다. 콜백은 새로운 규칙 또는 업데이트된 규칙의 ARN을 반환합니다.

```
// Load the AWS SDK for Node.js
```

```
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create CloudWatchEvents service object
var cwevents = new AWS.CloudWatchEvents({ apiVersion: "2015-10-07" });

var params = {
  Name: "DEMO_EVENT",
  RoleArn: "IAM_ROLE_ARN",
  ScheduleExpression: "rate(5 minutes)",
  State: "ENABLED",
};

cwevents.putRule(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.RuleArn);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node cwe_putrule.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

AWS Lambda 함수 대상 추가

파일 이름이 `cwe_puttargets.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. CloudWatch Events에 액세스하려면 `AWS.CloudWatchEvents` 서비스 객체를 생성합니다. 생성한 Lambda 함수의 ARN을 포함하여 대상에 연결할 규칙을 지정하는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. `putTargets` 서비스 객체의 `AWS.CloudWatchEvents` 메서드를 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create CloudWatchEvents service object
```

```
var cwevents = new AWS.CloudWatchEvents({ apiVersion: "2015-10-07" });

var params = {
  Rule: "DEMO_EVENT",
  Targets: [
    {
      Arn: "LAMBDA_FUNCTION_ARN",
      Id: "myCloudWatchEventsTarget",
    },
  ],
};

cwevents.putTargets(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node cwe_puttargets.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

이벤트 전송

파일 이름이 `cwe_putevents.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. CloudWatch Events에 액세스하려면 `AWS.CloudWatchEvents` 서비스 객체를 생성합니다. 이벤트를 전송하는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. 이벤트마다 해당 이벤트의 소스, 해당 이벤트의 영향을 받은 모든 리소스의 ARN, 해당 이벤트의 세부 정보를 포함시킵니다. `putEvents` 서비스 객체의 `AWS.CloudWatchEvents` 메서드를 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create CloudWatchEvents service object
var cwevents = new AWS.CloudWatchEvents({ apiVersion: "2015-10-07" });
```

```
var params = {
  Entries: [
    {
      Detail: '{ "key1": "value1", "key2": "value2" }',
      DetailType: "appRequestSubmitted",
      Resources: ["RESOURCE_ARN"],
      Source: "com.company.app",
    },
  ],
};

cwevents.putEvents(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.Entries);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node cwe_putevents.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

Amazon CloudWatch Logs 구독 필터 사용

이 Node.js 코드 예제는 다음을 보여 줍니다.

- CloudWatch Logs에서 로그 이벤트에 대한 필터를 생성하고 삭제하는 방법

시나리오

구독을 사용하면 CloudWatch Logs의 로그 이벤트 실시간 피드에 액세스할 수 있으며 사용자 지정 처리, 분석 또는 다른 시스템에 로드하기 위해 해당 피드를 Amazon Kinesis 스트림 또는 AWS Lambda와

같은 다른 서비스에 전달할 수 있습니다. 구독 필터는 AWS 리소스에 전달되는 로그 이벤트를 필터링하는 데 사용할 패턴을 정의합니다.

이 예제에서는 일련의 Node.js 모듈을 사용하여 CloudWatch Logs에서 구독 필터를 나열, 생성 및 삭제합니다. 로그 이벤트의 대상은 Lambda 함수입니다. Node.js 모듈은 SDK for JavaScript로 `CloudWatchLogs` 클라이언트 클래스의 다음 메서드를 사용하여 구독 필터를 관리합니다.

- `putSubscriptionFilters`
- `describeSubscriptionFilters`
- `deleteSubscriptionFilter`

자세한 내용은 Amazon CloudWatch Logs 사용 설명서의 구독을 통한 로그 데이터 실시간 처리를 참조하세요.

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 이러한 작업들을 완료해야 합니다.

- Node.js를 설치합니다. Node.js 설치에 대한 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 공유 자격 증명 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.
- Lambda 함수를 로그 이벤트의 대상으로 생성합니다. 이 함수의 ARN을 사용해야 합니다. Lambda 함수 설정에 대한 자세한 내용은 Amazon CloudWatch Logs 사용 설명서의 AWS Lambda 구독 필터를 참조하세요.
- 생성한 Lambda 함수를 간접 호출할 권한을 부여하고 CloudWatch Logs에 대한 모든 액세스 권한을 부여하는 정책이 있는 IAM 역할을 생성하거나, Lambda 함수에 대해 생성한 실행 역할에 다음 정책을 적용합니다. IAM 역할 생성에 관한 자세한 내용은 IAM 사용 설명서의 AWS 서비스에 대한 권한을 위임할 역할 생성 섹션을 참조하세요.

IAM 역할을 생성할 때 다음 역할 정책을 사용합니다.

JSON

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

```
            "Effect": "Allow",
            "Action": [
               "logs:CreateLogGroup",
               "logs:CreateLogStream",
               "logs:PutLogEvents"
            ],
            "Resource": "arn:aws:logs:*:*:*"
         },
         {
            "Effect": "Allow",
            "Action": [
               "lambda:InvokeFunction"
            ],
            "Resource": [
               "*"
            ]
         }
      ]
}
```

기존 구독 필터 설명

파일 이름이 `cwl_describesubscriptionfilters.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. CloudWatch Logs에 액세스하려면 `AWS.CloudWatchLogs` 서비스 객체를 생성합니다. 로그 그룹의 이름 및 설명할 최대 필터 수를 포함하여 기존 필터를 설명하는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. `describeSubscriptionFilters` 메서드를 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the CloudWatchLogs service object
var cwl = new AWS.CloudWatchLogs({ apiVersion: "2014-03-28" });

var params = {
  logGroupName: "GROUP_NAME",
  limit: 5,
};

cwl.describeSubscriptionFilters(params, function (err, data) {
```

```
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.subscriptionFilters);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node cwl_describesubscriptionfilters.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

구독 필터 생성

파일 이름이 `cwl_putsubscriptionfilter.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. CloudWatch Logs에 액세스하려면 `AWS.CloudWatchLogs` 서비스 객체를 생성합니다. 대상 Lambda 함수의 ARN, 필터의 이름, 필터링을 위한 문자열 패턴, 로그 그룹의 이름 등을 포함하여 필터를 생성하는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. `putSubscriptionFilters` 메서드를 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the CloudWatchLogs service object
var cwl = new AWS.CloudWatchLogs({ apiVersion: "2014-03-28" });

var params = {
  destinationArn: "LAMBDA_FUNCTION_ARN",
  filterName: "FILTER_NAME",
  filterPattern: "ERROR",
  logGroupName: "LOG_GROUP",
};

cwl.putSubscriptionFilter(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
```

```
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node cwl_putsubscriptionfilter.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

구독 필터 삭제

파일 이름이 `cwl_deletesubscriptionfilters.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. CloudWatch Logs에 액세스하려면 `AWS.CloudWatchLogs` 서비스 객체를 생성합니다. 필터 및 로그 그룹의 이름을 포함하여 필터를 삭제하는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. `deleteSubscriptionFilters` 메서드를 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the CloudWatchLogs service object
var cwl = new AWS.CloudWatchLogs({ apiVersion: "2014-03-28" });

var params = {
  filterName: "FILTER",
  logGroupName: "LOG_GROUP",
};

cwl.deleteSubscriptionFilter(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

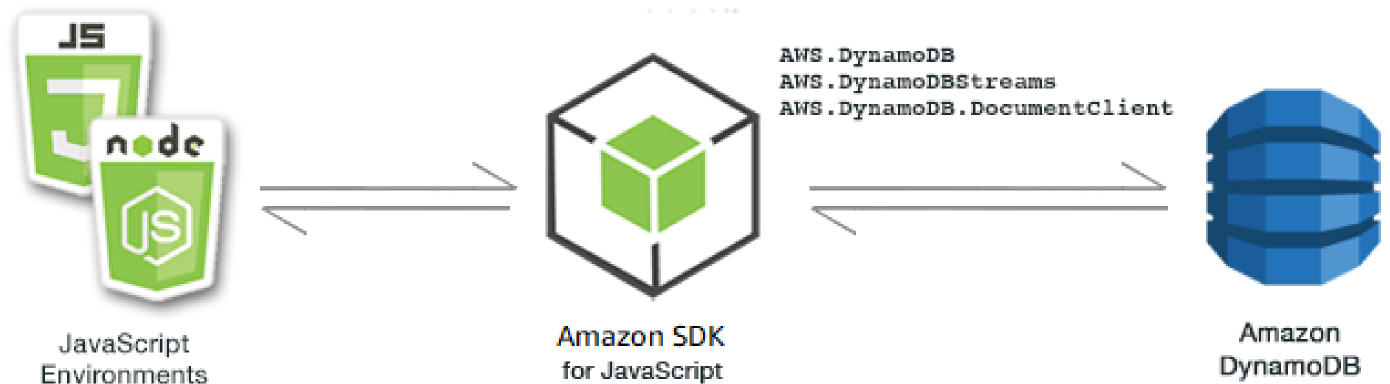
예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node cwl_deletesubscriptionfilter.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

Amazon DynamoDB 예제

Amazon DynamoDB는 문서 모델과 키-값 스토어 모델을 모두 지원하는 완전 관리형 NoSQL 클라우드 데이터베이스입니다. 전용 데이터베이스 서버를 프로비저닝하거나 유지 관리할 필요 없이 데이터에 대해 스키마 없는 테이블을 생성합니다.



DynamoDB용 JavaScript API는 `AWS.DynamoDB`, `AWS.DynamoDBStreams` 및 `AWS.DynamoDB.DocumentClient` 클라이언트 클래스를 통해 노출됩니다. DynamoDB 클라이언트 클래스 사용에 대한 자세한 내용은 API 참조의 [`Class: AWS.DynamoDB`], [`Class: AWS.DynamoDBStreams`], [`Class: AWS.DynamoDB.DocumentClient`] 섹션을 참조하세요.

주제

- DynamoDB에서 테이블 생성 및 사용
- DynamoDB에서 단일 항목 읽기 및 쓰기
- DynamoDB에서 배치로 항목 읽기 및 쓰기
- DynamoDB 테이블 쿼리 및 스캔
- DynamoDB 문서 클라이언트 사용

DynamoDB에서 테이블 생성 및 사용

이 Node.js 코드 예제는 다음을 보여 줍니다.

- DynamoDB에서 데이터를 저장하고 검색하는 데 사용되는 테이블을 생성하고 관리하는 방법

시나리오

다른 데이터베이스 시스템과 마찬가지로 DynamoDB는 데이터를 테이블에 저장합니다. DynamoDB 테이블은 행과 유사한 항목으로 구성되는 데이터의 모음입니다. DynamoDB에서 데이터를 저장하거나 데이터에 액세스하려면 테이블을 생성하고 테이블로 작업합니다.

이 예에서는 일련의 Node.js 모듈을 사용하여 DynamoDB 테이블의 기본 작업을 수행합니다. 이 코드는 SDK for JavaScript에서 `AWS.DynamoDB` 클라이언트 클래스의 다음 메서드를 사용하여 테이블을 생성하고 테이블로 작업합니다.

- [createTable]
- [listTables]
- [describeTable]
- [deleteTable]

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 다음 작업을 완료합니다.

- Node.js를 설치합니다. 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 공유 자격 증명 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.

표 생성

파일 이름이 `ddb_createtable.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. DynamoDB에 액세스하려면 `AWS.DynamoDB` 서비스 객체를 생성합니다. 테이블을 생성하는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. 이 예제에서는 각 속성의 이름과 데이터 형식, 키 스키마, 테이블의 이름, 프로비저닝할 처리량의 단위가 포함됩니다. DynamoDB 객체의 `createTable` 메서드를 직접 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });
```

```
var params = {
  AttributeDefinitions: [
    {
      AttributeName: "CUSTOMER_ID",
      AttributeType: "N",
    },
    {
      AttributeName: "CUSTOMER_NAME",
      AttributeType: "S",
    },
  ],
  KeySchema: [
    {
      AttributeName: "CUSTOMER_ID",
      KeyType: "HASH",
    },
    {
      AttributeName: "CUSTOMER_NAME",
      KeyType: "RANGE",
    },
  ],
  ProvisionedThroughput: {
    ReadCapacityUnits: 1,
    WriteCapacityUnits: 1,
  },
  TableName: "CUSTOMER_LIST",
  StreamSpecification: {
    StreamEnabled: false,
  },
};

// Call DynamoDB to create the table
ddb.createTable(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Table Created", data);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node ddb_createtable.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

테이블 목록 조회

파일 이름이 ddb_listtables.js인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. DynamoDB에 액세스하려면 AWS.DynamoDB 서비스 객체를 생성합니다. 테이블을 나열하는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다, 이 예제에서는 나열되는 테이블 수를 10으로 제한합니다. DynamoDB 객체의 listTables 메서드를 직접 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

// Call DynamoDB to retrieve the list of tables
ddb.listTables({ Limit: 10 }, function (err, data) {
  if (err) {
    console.log("Error", err.code);
  } else {
    console.log("Table names are ", data.TableNames);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node ddb_listtables.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

테이블 설명

파일 이름이 ddb_describetable.js인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. DynamoDB에 액세스하려면 AWS.DynamoDB 서비스 객체를 생성합니다. 테이블을 설명하는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. 이 예제에서는 명령줄 파라미터로 제공되는 테이블의 이름이 포함됩니다. DynamoDB 객체의 describeTable 메서드를 직접 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  TableName: process.argv[2],
};

// Call DynamoDB to retrieve the selected table descriptions
ddb.describeTable(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.Table.KeySchema);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node ddb_describetable.js TABLE_NAME
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

테이블 삭제

파일 이름이 `ddb_deletetable.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. DynamoDB에 액세스하려면 `AWS.DynamoDB` 서비스 객체를 생성합니다. 테이블을 삭제하는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. 이 예에서는 명령줄 파라미터로 제공되는 테이블의 이름을 포함합니다. DynamoDB 객체의 `deleteTable` 메서드를 직접 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });
```

```
var params = {
  TableName: process.argv[2],
};

// Call DynamoDB to delete the specified table
ddb.deleteTable(params, function (err, data) {
  if (err && err.code === "ResourceNotFoundException") {
    console.log("Error: Table not found");
  } else if (err && err.code === "ResourceInUseException") {
    console.log("Error: Table in use");
  } else {
    console.log("Success", data);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node ddb_deletetable.js TABLE_NAME
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

DynamoDB에서 단일 항목 읽기 및 쓰기

이 Node.js 코드 예제는 다음을 보여 줍니다.

- DynamoDB 테이블에서 항목을 추가하는 방법
- DynamoDB 테이블에서 항목을 검색하는 방법
- DynamoDB 테이블에서 항목을 삭제하는 방법

시나리오

이 예에서는 일련의 Node.js 모듈에서 `AWS.DynamoDB` 클라이언트 클래스의 다음 메서드를 사용하여 DynamoDB 테이블에서 항목 하나를 읽고 씁니다.

- putItem

- getItem
- deleteItem

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 다음 작업을 완료합니다.

- Node.js를 설치합니다. 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 공유 자격 증명 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.
- 액세스할 수 있는 항목이 포함된 DynamoDB 테이블을 생성합니다. DynamoDB 테이블 생성에 관한 자세한 내용은 DynamoDB에서 테이블 생성 및 사용 섹션을 참조하세요.

항목 쓰기

파일 이름이 `ddb_putitem.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. DynamoDB에 액세스하려면 `AWS.DynamoDB` 서비스 객체를 생성합니다. 항목을 추가하는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. 이 예제에서는 테이블의 이름 및 설정할 속성과 각 속성의 값을 정의하는 맵이 포함됩니다. DynamoDB 객체의 `putItem` 메서드를 직접 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  TableName: "CUSTOMER_LIST",
  Item: {
    CUSTOMER_ID: { N: "001" },
    CUSTOMER_NAME: { S: "Richard Roe" },
  },
};

// Call DynamoDB to add the item to the table
ddb.putItem(params, function (err, data) {
  if (err) {
```

```
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node ddb_putitem.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

항목 가져오기

파일 이름이 `ddb_getitem.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. DynamoDB에 액세스하려면 `AWS.DynamoDB` 서비스 객체를 생성합니다. 가져올 항목을 식별하려면 테이블의 해당 항목에 대한 기본 키의 값을 제공해야 합니다. 기본적으로 `getItem` 메서드는 항목에 정의된 모든 속성 값을 반환합니다. 가능한 모든 속성 값의 일부만 가져오려면 프로젝션 표현식을 지정합니다.

항목을 가져오는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. 이 예제에서는 테이블의 이름, 가져오는 항목에 대한 키의 이름과 값, 검색할 항목 속성을 식별하는 프로젝션 표현식이 포함됩니다. DynamoDB 객체의 `getItem` 메서드를 직접 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  TableName: "TABLE",
  Key: {
    KEY_NAME: { N: "001" },
  },
  ProjectionExpression: "ATTRIBUTE_NAME",
};

// Call DynamoDB to read the item from the table
```

```
ddb.getItem(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.Item);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node ddb_getitem.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

항목 삭제

파일 이름이 `ddb_deleteitem.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. DynamoDB에 액세스하려면 `AWS.DynamoDB` 서비스 객체를 생성합니다. 항목을 삭제하는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. 이 예에서는 테이블의 이름과 삭제하려는 항목의 키 이름 및 값을 포함합니다. DynamoDB 객체의 `deleteItem` 메서드를 직접 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  TableName: "TABLE",
  Key: {
    KEY_NAME: { N: "VALUE" },
  },
};

// Call DynamoDB to delete the item from the table
ddb.deleteItem(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
```

```
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node ddb_deleteitem.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

DynamoDB에서 배치로 항목 읽기 및 쓰기

이 Node.js 코드 예제는 다음을 보여 줍니다.

- DynamoDB 테이블에서 항목의 배치를 읽고 쓰는 방법

시나리오

이 예제에서는 일련의 Node.js 모듈을 사용하여 항목의 배치를 DynamoDB table 테이블에 넣고 항목의 배치를 읽습니다. 이 코드는 SDK for JavaScript에서 DynamoDB 클라이언트 클래스의 다음 메서드를 사용하여 배치 읽기 및 쓰기 작업을 수행합니다.

- batchGetItem
- batchWriteItem

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 다음 작업을 완료합니다.

- Node.js를 설치합니다. 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 공유 자격 증명 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.
- 액세스할 수 있는 항목이 포함된 DynamoDB 테이블을 생성합니다. DynamoDB 테이블 생성에 관한 자세한 내용은 DynamoDB에서 테이블 생성 및 사용 섹션을 참조하세요.

배치로 항목 읽기

파일 이름이 `ddb_batchgetitem.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. DynamoDB에 액세스하려면 `AWS.DynamoDB` 서비스 객체를 생성합니다. 항목의 배치를 가져오는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. 이 예제에서는 읽을 하나 이상 테이블의 이름, 각 테이블에서 읽을 키의 값, 반환할 속성을 지정하는 프로젝션 표현식이 포함됩니다. DynamoDB 객체의 `batchGetItem` 메서드를 직접 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  RequestItems: {
    TABLE_NAME: {
      Keys: [
        { KEY_NAME: { N: "KEY_VALUE_1" } },
        { KEY_NAME: { N: "KEY_VALUE_2" } },
        { KEY_NAME: { N: "KEY_VALUE_3" } },
      ],
      ProjectionExpression: "KEY_NAME, ATTRIBUTE",
    },
  },
};

ddb.batchGetItem(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    data.Responses.TABLE_NAME.forEach(function (element, index, array) {
      console.log(element);
    });
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node ddb_batchgetitem.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

배치로 항목 쓰기

파일 이름이 `ddb_batchwriteitem.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. DynamoDB에 액세스하려면 `AWS.DynamoDB` 서비스 객체를 생성합니다. 항목의 배치를 가져오는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. 이 예제에서는 항목을 쓰려는 테이블, 각 항목에 대해 쓰려는 키, 속성 및 해당 값이 이러한 객체입니다. DynamoDB 객체의 `batchWriteItem` 메서드를 직접 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  RequestItems: {
    TABLE_NAME: [
      {
        PutRequest: {
          Item: {
            KEY: { N: "KEY_VALUE" },
            ATTRIBUTE_1: { S: "ATTRIBUTE_1_VALUE" },
            ATTRIBUTE_2: { N: "ATTRIBUTE_2_VALUE" },
          },
        },
      },
      {
        PutRequest: {
          Item: {
            KEY: { N: "KEY_VALUE" },
            ATTRIBUTE_1: { S: "ATTRIBUTE_1_VALUE" },
            ATTRIBUTE_2: { N: "ATTRIBUTE_2_VALUE" },
          },
        },
      },
    ],
  },
};
```

```
ddb.batchWriteItem(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node ddb_batchwriteitem.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

DynamoDB 테이블 쿼리 및 스캔

이 Node.js 코드 예제는 다음을 보여 줍니다.

- DynamoDB 테이블에서 항목을 쿼리하고 스캔하는 방법

시나리오

쿼리는 기본 키 속성 값만 사용하여 테이블 또는 보조 인덱스에서 항목을 찾습니다. 검색할 파티션 키 이름과 값을 제공해야 합니다. 정렬 키 이름과 값도 제공할 수 있으며, 비교 연산자를 사용하여 검색 결과를 좁힐 수 있습니다. 스캔은 지정된 테이블에서 모든 항목을 확인하여 항목을 찾습니다.

이 예에서는 일련의 Node.js 모듈을 사용하여 DynamoDB 테이블에서 검색하려는 항목을 하나 이상 식별합니다. 이 코드는 SDK for JavaScript에서 DynamoDB 클라이언트 클래스의 다음 메서드를 사용하여 테이블을 쿼리하고 스캔합니다.

- query
- 스캔

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 다음 작업을 완료합니다.

- Node.js를 설치합니다. 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 공유 자격 증명 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.
- 액세스할 수 있는 항목이 포함된 DynamoDB 테이블을 생성합니다. DynamoDB 테이블 생성에 관한 자세한 내용은 DynamoDB에서 테이블 생성 및 사용 섹션을 참조하세요.

테이블 쿼리

이 예제에서는 비디오 시리즈에 대한 에피소드 정보가 포함된 테이블을 쿼리하여 지정된 어구가 자막에 포함되어 있는 에피소드 9 이후 두 번째 시즌 에피소드의 에피소드 제목과 자막을 반환합니다.

파일 이름이 `ddb_query.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. DynamoDB에 액세스하려면 `AWS.DynamoDB` 서비스 객체를 생성합니다. 테이블을 쿼리하는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. 이 예제에서는 테이블 이름, 쿼리에 필요한 `ExpressionAttributeValues`, 해당 값을 사용하여 쿼리에서 반환되는 항목을 정의하는 `KeyConditionExpression`, 각 항목에 대해 반환할 속성 값의 이름이 포함됩니다. DynamoDB 객체의 `query` 메서드를 직접 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create DynamoDB service object
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

var params = {
  ExpressionAttributeValues: {
    ":s": { N: "2" },
    ":e": { N: "09" },
    ":topic": { S: "PHRASE" },
  },
  KeyConditionExpression: "Season = :s and Episode > :e",
  ProjectionExpression: "Episode, Title, Subtitle",
  FilterExpression: "contains (Subtitle, :topic)",
  TableName: "EPISODES_TABLE",
```

```
};

ddb.query(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    //console.log("Success", data.Items);
    data.Items.forEach(function (element, index, array) {
      console.log(element.Title.S + " (" + element.Subtitle.S + ")");
    });
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node ddb_query.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

테이블 스캔

파일 이름이 `ddb_scan.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. DynamoDB에 액세스하려면 `AWS.DynamoDB` 서비스 객체를 생성합니다. 테이블에서 항목을 스캔하는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. 이 예제에서는 테이블의 이름, 일치하는 각 항목에 대해 반환할 속성 값의 목록, 결과 집합을 필터링하여 지정된 어구가 포함된 항목을 찾는 표현식이 포함됩니다. DynamoDB 객체의 `scan` 메서드를 직접 호출합니다.

```
// Load the AWS SDK for Node.js.
var AWS = require("aws-sdk");
// Set the AWS Region.
AWS.config.update({ region: "REGION" });

// Create DynamoDB service object.
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });

const params = {
  // Specify which items in the results are returned.
  FilterExpression: "Subtitle = :topic AND Season = :s AND Episode = :e",
  // Define the expression attribute value, which are substitutes for the values you
 want to compare.
  ExpressionAttributeValues: {
```

```
    ":topic": { S: "SubTitle2" },
    ":s": { N: 1 },
    ":e": { N: 2 },
  },
  // Set the projection expression, which are the attributes that you want.
  ProjectionExpression: "Season, Episode, Title, Subtitle",
  TableName: "EPISODES_TABLE",
};

ddb.scan(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
    data.Items.forEach(function (element, index, array) {
      console.log(
        "printing",
        element.Title.S + " (" + element.Subtitle.S + ")"
      );
    });
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node ddb_scan.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

DynamoDB 문서 클라이언트 사용

이 Node.js 코드 예제는 다음을 보여 줍니다.

- 문서 클라이언트를 사용하여 DynamoDB 테이블에 액세스하는 방법

시나리오

DynamoDB 문서 클라이언트는 속성 값의 개념을 추상화하여 항목 작업을 간소화합니다. 이 추상화는 입력 파라미터로 제공되는 기본 JavaScript 유형에 주석을 달고, 주석이 달린 응답 데이터를 기본 JavaScript 유형으로 변환합니다.

DynamoDB 문서 클라이언트 클래스에 대한 자세한 내용은 API 참조의 `AWS.DynamoDB.DocumentClient` 섹션을 참조하세요. Amazon DynamoDB를 사용한 프로그래밍에 관한 자세한 내용은 Amazon DynamoDB 개발자 안내서의 DynamoDB를 사용한 프로그래밍을 참조하세요.

이 예제에서는 일련의 Node.js 모듈에서 문서 클라이언트를 사용하여 DynamoDB 테이블에 대한 기본 작업을 수행합니다. 이 코드는 SDK for JavaScript에서 DynamoDB 문서 클라이언트 클래스의 다음 메서드를 사용하여 테이블을 쿼리하고 스캔합니다.

- get
- put
- update
- query
- delete

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 다음 작업을 완료합니다.

- Node.js를 설치합니다. 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 공유 자격 증명 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.
- 액세스할 수 있는 항목이 포함된 DynamoDB 테이블을 생성합니다. SDK for JavaScript를 사용하여 DynamoDB 테이블을 생성하는 방법에 관한 자세한 내용은 DynamoDB에서 테이블 생성 및 사용 단원을 참조하세요. DynamoDB 콘솔을 사용하여 테이블을 생성할 수도 있습니다.

테이블에서 항목 가져오기

파일 이름이 `ddbdoc_get.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. DynamoDB에 액세스하려면 `AWS.DynamoDB.DocumentClient` 객체를 생성합니다. 테이블에서 항

목을 가져오는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. 이 예제에서는 테이블의 이름, 해당 테이블에 있는 해시 키의 이름, 가져올 항목에 대한 해시 키의 값이 포함됩니다. DynamoDB 문서 클라이언트의 `get` 메서드를 직접 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create DynamoDB document client
var docClient = new AWS.DynamoDB.DocumentClient({ apiVersion: "2012-08-10" });

var params = {
  TableName: "EPISODES_TABLE",
  Key: { KEY_NAME: VALUE },
};

docClient.get(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.Item);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node ddbdoc_get.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

테이블에 항목 넣기

파일 이름이 `ddbdoc_put.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. DynamoDB에 액세스하려면 `AWS.DynamoDB.DocumentClient` 객체를 생성합니다. 항목을 테이블에 쓰는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. 이 예제에서는 테이블의 이름, 해시 키 및 값을 포함하여 추가하거나 업데이트할 항목에 대한 설명, 항목에서 설정할 속성의 이름과 값이 포함됩니다. DynamoDB 문서 클라이언트의 `put` 메서드를 직접 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
```

```
// Set the region
AWS.config.update({ region: "REGION" });

// Create DynamoDB document client
var docClient = new AWS.DynamoDB.DocumentClient({ apiVersion: "2012-08-10" });

var params = {
  TableName: "TABLE",
  Item: {
    HASHKEY: VALUE,
    ATTRIBUTE_1: "STRING_VALUE",
    ATTRIBUTE_2: VALUE_2,
  },
};

docClient.put(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node ddbdoc_put.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

테이블에서 항목 업데이트

파일 이름이 `ddbdoc_update.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. DynamoDB에 액세스하려면 `AWS.DynamoDB.DocumentClient` 객체를 생성합니다. 항목을 테이블에 쓰는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. 이 예제에서는 테이블의 이름, 업데이트할 항목의 키, `ExpressionAttributeValues` 파라미터에서 값을 할당하는 토큰을 사용하여 업데이트할 항목의 속성을 정의하는 `UpdateExpressions` 집합이 포함됩니다. DynamoDB 문서 클라이언트의 `update` 메서드를 직접 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });
```

```
// Create DynamoDB document client
var docClient = new AWS.DynamoDB.DocumentClient({ apiVersion: "2012-08-10" });

// Create variables to hold numeric key values
var season = SEASON_NUMBER;
var episode = EPISODES_NUMBER;

var params = {
  TableName: "EPISODES_TABLE",
  Key: {
    Season: season,
    Episode: episode,
  },
  UpdateExpression: "set Title = :t, Subtitle = :s",
  ExpressionAttributeValues: {
    ":t": "NEW_TITLE",
    ":s": "NEW_SUBTITLE",
  },
};

docClient.update(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node ddbdoc_update.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

테이블 쿼리

이 예제에서는 비디오 시리즈에 대한 에피소드 정보가 포함된 테이블을 쿼리하여 지정된 어구가 자막에 포함되어 있는 에피소드 9 이후 두 번째 시즌 에피소드의 에피소드 제목과 자막을 반환합니다.

파일 이름이 `ddbdoc_query.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. DynamoDB에 액세스하려면 `AWS.DynamoDB.DocumentClient` 객체를 생성합니다. 테이블을 쿼리

하는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. 이 예제에서는 테이블 이름, 쿼리에 필요한 `ExpressionAttributeValues`, 해당 값을 사용하여 쿼리에서 반환되는 항목을 정의하는 `KeyConditionExpression`이 포함됩니다. DynamoDB 문서 클라이언트의 `query` 메서드를 직접 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create DynamoDB document client
var docClient = new AWS.DynamoDB.DocumentClient({ apiVersion: "2012-08-10" });

var params = {
  ExpressionAttributeValues: {
    ":s": 2,
    ":e": 9,
    ":topic": "PHRASE",
  },
  KeyConditionExpression: "Season = :s and Episode > :e",
  FilterExpression: "contains (Subtitle, :topic)",
  TableName: "EPISODES_TABLE",
};

docClient.query(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.Items);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node ddbdoc_query.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

테이블에서 항목 삭제

파일 이름이 `ddbdoc_delete.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. DynamoDB에 액세스하려면 `AWS.DynamoDB.DocumentClient` 객체를 생성합니다. 테이블에서 항

목을 삭제하는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. 이 예제에서는 테이블의 이름, 삭제할 항목에 대한 해시 키의 이름과 값이 포함됩니다. DynamoDB 문서 클라이언트의 `delete` 메서드를 직접 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create DynamoDB document client
var docClient = new AWS.DynamoDB.DocumentClient({ apiVersion: "2012-08-10" });

var params = {
  Key: {
    HASH_KEY: VALUE,
  },
  TableName: "TABLE",
};

docClient.delete(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node ddbdoc_delete.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

Amazon EC2 예제

Amazon Elastic Compute Cloud(Amazon EC2)는 클라우드에서 가상 서버 호스팅을 제공하는 웹 서비스입니다. 이 서비스는 크기 조정 가능한 컴퓨팅 파워를 제공하여 개발자가 더 쉽게 웹 규모 클라우드 컴퓨팅을 수행할 수 있도록 설계되었습니다.

Amazon EC2용 JavaScript API는 `AWS.EC2` 클라이언트 클래스를 통해 노출됩니다. Amazon EC2 클라이언트 클래스 사용에 대한 자세한 내용은 API 참조의 `Class: AWS.EC2` 섹션을 참조하세요.

주제

- Amazon EC2 인스턴스 생성
- Amazon EC2 인스턴스 관리
- Amazon EC2 키 페어로 작업
- Amazon EC2에서 리전 및 가용 영역 사용
- Amazon EC2의 보안 그룹 작업
- Amazon EC2에서 탄력적 IP 주소 사용

Amazon EC2 인스턴스 생성

이 Node.js 코드 예제는 다음을 보여 줍니다.

- 퍼블릭 Amazon Machine Image(AMI)에서 Amazon EC2 인스턴스를 생성하는 방법
- 새 Amazon EC2 인스턴스를 생성하고 해당 인스턴스에 태그를 할당하는 방법

예제 소개

이 예제에서는 Node.js 모듈을 사용하여 Amazon EC2 인스턴스를 생성하고 해당 인스턴스에 키 페어와 태그를 모두 할당합니다. 이 코드는 Amazon EC2 클라이언트 클래스의 다음 메서드를 사용하여 인스턴스를 생성하고 인스턴스에 태그를 지정할 수 있도록 SDK for JavaScript를 사용합니다.

- `runInstances`
- `createTags`

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 다음 작업을 완료합니다.

- Node.js를 설치합니다. 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 공유 자격 증명 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.
- 키 페어를 생성합니다. 세부 정보는 Amazon EC2 키 페어로 작업을 참조하세요. 이 예제에서는 키 페어의 이름을 사용합니다.

인스턴스 생성 및 태그 지정

파일 이름이 `ec2_createinstances.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다.

할당할 키 페어의 이름과 실행할 AMI의 ID를 포함하여 AWS.EC2 클라이언트 클래스의 `runInstances` 메서드에 대한 파라미터를 전달할 객체를 생성합니다. `runInstances` 메서드를 호출하려면 파라미터를 전달하는 Amazon EC2 서비스 객체를 호출하기 위한 promise를 생성합니다. 그런 다음 promise 콜백에서 response를 처리합니다.

다음 코드로 Name 태그를 새 인스턴스에 추가하면 Amazon EC2 콘솔이 새 인스턴스를 인식하고 인스턴스 목록의 이름 필드에 표시합니다. 한 인스턴스에 최대 50개의 태그를 추가할 수 있으며, `createTags` 메서드에 대한 단일 호출에서 모두 추가할 수 있습니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Load credentials and set region from JSON file
AWS.config.update({ region: "REGION" });

// Create EC2 service object
```

```
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

// AMI is amzn-ami-2011.09.1.x86_64-ebs
var instanceParams = {
  ImageId: "AMI_ID",
  InstanceType: "t2.micro",
  KeyName: "KEY_PAIR_NAME",
  MinCount: 1,
  MaxCount: 1,
};

// Create a promise on an EC2 service object
var instancePromise = new AWS.EC2({ apiVersion: "2016-11-15" })
  .runInstances(instanceParams)
  .promise();

// Handle promise's fulfilled/rejected states
instancePromise
  .then(function (data) {
    console.log(data);
    var instanceId = data.Instances[0].InstanceId;
    console.log("Created instance", instanceId);
    // Add tags to the instance
    tagParams = {
      Resources: [instanceId],
      Tags: [
        {
          Key: "Name",
          Value: "SDK Sample",
        },
      ],
    };
    // Create a promise on an EC2 service object
    var tagPromise = new AWS.EC2({ apiVersion: "2016-11-15" })
      .createTags(tagParams)
      .promise();
    // Handle promise's fulfilled/rejected states
    tagPromise
      .then(function (data) {
        console.log("Instance tagged");
      })
      .catch(function (err) {
        console.error(err, err.stack);
      });
```

```
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node ec2_createinstances.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

Amazon EC2 인스턴스 관리

이 Node.js 코드 예제는 다음을 보여 줍니다.

- Amazon EC2 인스턴스에 대한 기본 정보를 검색하는 방법
- Amazon EC2 인스턴스의 세부 모니터링을 시작하고 중지하는 방법
- Amazon EC2 인스턴스를 시작하고 중지하는 방법
- Amazon EC2 인스턴스 재부팅 방법

시나리오

이 예제에서는 일련의 Node.js 모듈을 사용하여 여러 가지 기본 인스턴스 관리 작업을 수행합니다. Node.js 모듈은 SDK for JavaScript로 다음의 Amazon EC2 클라이언트 클래스 메서드를 사용하여 인스턴스를 관리합니다.

- `describeInstances`
- `monitorInstances`
- `unmonitorInstances`
- `startInstances`
- `stopInstances`

- rebootInstances

Amazon EC2 인스턴스 수명 주기에 대한 자세한 내용은 Amazon EC2 사용 설명서의 인스턴스 수명 주기를 참조하세요.

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 다음 작업을 완료합니다.

- Node.js를 설치합니다. Node.js 설치에 대한 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 공유 자격 증명 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.
- Amazon EC2 인스턴스 생성 Amazon EC2 인스턴스 생성에 대한 자세한 내용은 Amazon EC2 사용 설명서의 Amazon EC2 인스턴스 또는 Amazon EC2 사용 설명서의 Amazon EC2 인스턴스를 참조하세요.

인스턴스 설명

파일 이름이 `ec2_describeinstances.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. Amazon EC2에 액세스하려면 `AWS.EC2` 서비스 객체를 생성합니다. Amazon EC2 서비스 객체의 `describeInstances` 메서드를 직접 호출하여 인스턴스에 대한 세부 설명을 검색합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

var params = {
  DryRun: false,
};

// Call EC2 to retrieve policy for selected bucket
ec2.describeInstances(params, function (err, data) {
  if (err) {
    console.log("Error", err.stack);
  } else {
```

```
    console.log("Success", JSON.stringify(data));
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node ec2_describeinstances.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

인스턴스 모니터링 관리

파일 이름이 `ec2_monitorinstances.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. Amazon EC2에 액세스하려면 `AWS.EC2` 서비스 객체를 생성합니다. 모니터링을 제어하려는 인스턴스의 인스턴스 ID를 추가합니다.

명령줄 인수의 값(`ON` 또는 `OFF`)을 기반으로 Amazon EC2 서비스 객체의 `monitorInstances` 메서드를 호출하여 지정된 인스턴스에 대한 세부 모니터링을 시작하거나 `unmonitorInstances` 메서드를 직접 호출합니다. 이러한 인스턴스에 대한 모니터링을 변경하려고 시도하기 전에 `DryRun` 파라미터를 사용하여 인스턴스 모니터링을 변경할 권한이 있는지 여부를 테스트합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

var params = {
  InstanceIds: ["INSTANCE_ID"],
  DryRun: true,
};

if (process.argv[2].toUpperCase() === "ON") {
  // Call EC2 to start monitoring the selected instances
  ec2.monitorInstances(params, function (err, data) {
    if (err && err.code === "DryRunOperation") {
      params.DryRun = false;
      ec2.monitorInstances(params, function (err, data) {
        if (err) {
```

```
          console.log("Error", err);
        } else if (data) {
          console.log("Success", data.InstanceMonitorings);
        }
      });
    } else {
      console.log("You don't have permission to change instance monitoring.");
    }
  });
} else if (process.argv[2].toUpperCase() === "OFF") {
  // Call EC2 to stop monitoring the selected instances
  ec2.unmonitorInstances(params, function (err, data) {
    if (err && err.code === "DryRunOperation") {
      params.DryRun = false;
      ec2.unmonitorInstances(params, function (err, data) {
        if (err) {
          console.log("Error", err);
        } else if (data) {
          console.log("Success", data.InstanceMonitorings);
        }
      });
    } else {
      console.log("You don't have permission to change instance monitoring.");
    }
  });
}
```

예제를 실행하려면 명령줄에 다음을 입력하면서 ON을 지정하여 세부 모니터링을 시작하거나 OFF를 지정하여 모니터링을 중단합니다.

```
node ec2_monitorinstances.js ON
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

인스턴스 시작 및 중지

파일 이름이 `ec2_startstopinstances.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. Amazon EC2에 액세스하려면 `AWS.EC2` 서비스 객체를 생성합니다. 시작하거나 중지할 인스턴스의 인스턴스 ID를 추가합니다.

명령줄 인수의 값(START 또는 STOP)을 기반으로 Amazon EC2 서비스 객체의 `startInstances` 메서드를 호출하여 지정된 인스턴스를 시작하거나 `stopInstances` 메서드를 직접 호출하여 지정된 인

스턴스를 중지합니다. 선택한 인스턴스를 실제로 시작하거나 중지하려고 시도하기 전에 DryRun 파라미터를 사용하여 권한이 있는지 여부를 테스트합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

var params = {
  InstanceIds: [process.argv[3]],
  DryRun: true,
};

if (process.argv[2].toUpperCase() === "START") {
  // Call EC2 to start the selected instances
  ec2.startInstances(params, function (err, data) {
    if (err && err.code === "DryRunOperation") {
      params.DryRun = false;
      ec2.startInstances(params, function (err, data) {
        if (err) {
          console.log("Error", err);
        } else if (data) {
          console.log("Success", data.StartingInstances);
        }
      });
    } else {
      console.log("You don't have permission to start instances.");
    }
  });
} else if (process.argv[2].toUpperCase() === "STOP") {
  // Call EC2 to stop the selected instances
  ec2.stopInstances(params, function (err, data) {
    if (err && err.code === "DryRunOperation") {
      params.DryRun = false;
      ec2.stopInstances(params, function (err, data) {
        if (err) {
          console.log("Error", err);
        } else if (data) {
          console.log("Success", data.StoppingInstances);
        }
```

```
      });
    } else {
      console.log("You don't have permission to stop instances");
    }
  });
}
```

예제를 실행하려면 명령줄에 다음을 입력하면서 START를 지정하여 인스턴스를 시작하거나 STOP을 지정하여 인스턴스를 중지합니다.

```
node ec2_startstopinstances.js START INSTANCE_ID
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

인스턴스 재부팅

파일 이름이 `ec2_rebootinstances.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. Amazon EC2에 액세스하려면 Amazon EC2 서비스 객체를 생성합니다. 재부팅할 인스턴스의 인스턴스 ID를 추가합니다. `AWS.EC2` 서비스 객체의 `rebootInstances` 메서드를 호출하여 지정된 인스턴스를 재부팅합니다. 실제로 인스턴스를 재부팅하려고 시도하기 전에 `DryRun` 파라미터를 사용하여 이러한 인스턴스를 재부팅할 권한이 있는지 여부를 테스트합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

var params = {
  InstanceIds: ["INSTANCE_ID"],
  DryRun: true,
};

// Call EC2 to reboot instances
ec2.rebootInstances(params, function (err, data) {
  if (err && err.code === "DryRunOperation") {
    params.DryRun = false;
    ec2.rebootInstances(params, function (err, data) {
      if (err) {
        console.log("Error", err);
```

```
      } else if (data) {
        console.log("Success", data);
      }
    });
  } else {
    console.log("You don't have permission to reboot instances.");
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node ec2_rebootinstances.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

Amazon EC2 키 페어로 작업

이 Node.js 코드 예제는 다음을 보여 줍니다.

- 키 페어에 대한 정보를 검색하는 방법.
- Amazon EC2 인스턴스에 액세스하기 위해 키 페어를 생성하는 방법
- 기존 키 페어를 삭제하는 방법.

시나리오

Amazon EC2는 퍼블릭 키 암호화 기법을 사용하여 로그인 정보를 암호화 및 해독합니다. 퍼블릭 키 암호화 기법에서는 퍼블릭 키를 사용하여 데이터를 암호화한 후 수신자가 개인 키를 사용하여 해당 데이터를 해독합니다. 퍼블릭 키와 프라이빗 키를 키 페어라고 합니다.

이 예제에서는 일련의 Node.js 모듈을 사용하여 여러 가지 Amazon EC2 키 페어 관리 작업을 수행합니다. Node.js 모듈은 SDK for JavaScript로 Amazon EC2 클라이언트 클래스의 다음 메서드를 사용하여 인스턴스를 관리합니다

- `createKeyPair`
- `deleteKeyPair`

- [describeKeyPairs]

Amazon EC2 키 페어에 대한 자세한 내용은 Amazon EC2 사용 설명서의 [Amazon EC2 키 페어] 또는 Amazon EC2 사용 설명서의 [Amazon EC2 키 페어 및 Windows 인스턴스]를 참조하세요.

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 다음 작업을 완료합니다.

- Node.js를 설치합니다. Node.js 설치에 대한 자세한 내용은 [Node.js 웹 사이트]를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 공유 자격 증명 파일 제공에 대한 자세한 내용은 [공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드] 섹션을 참조하세요.

키 페어 설명

파일 이름이 `ec2_describekeypairs.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. Amazon EC2에 액세스하려면 `AWS.EC2` 서비스 객체를 생성합니다. `describeKeyPairs` 메서드가 모든 키 페어에 대한 설명을 반환하는 데 필요한 파라미터를 담을 비어 있는 JSON 객체를 생성합니다. JSON 파일에 포함된 파라미터의 `KeyName` 부분에 있는 키 페어의 이름 배열을 `describeKeyPairs` 메서드에 제공할 수도 있습니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

// Retrieve key pair descriptions; no params needed
ec2.describeKeyPairs(function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", JSON.stringify(data.KeyPairs));
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node ec2_describekeypairs.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

키 페어 만들기

각 키 페어에는 이름이 필요합니다. Amazon EC2는 사용자가 키 이름으로 지정한 이름에 퍼블릭 키를 연결합니다. 파일 이름이 `ec2_createkeypair.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. Amazon EC2에 액세스하려면 `AWS.EC2` 서비스 객체를 생성합니다. 키 페어의 이름을 지정할 JSON 파라미터를 생성한 다음 해당 파라미터를 전달하여 `createKeyPair` 메서드를 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

var params = {
  KeyName: "KEY_PAIR_NAME",
};

// Create the key pair
ec2.createKeyPair(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log(JSON.stringify(data));
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node ec2_createkeypair.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

키 페어 삭제

파일 이름이 `ec2_deletekeypair.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. Amazon EC2에 액세스하려면 `AWS.EC2` 서비스 객체를 생성합니다. 삭제하려는 키 페어의 이름을 지정할 JSON 파라미터를 생성합니다. 그런 다음 `deleteKeyPair` 메서드를 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

var params = {
  KeyName: "KEY_PAIR_NAME",
};

// Delete the key pair
ec2.deleteKeyPair(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Key Pair Deleted");
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node ec2_deletekeypair.js
```

이 샘플 코드는 [GitHub](https://github.com)에서 찾을 수 있습니다.

Amazon EC2에서 리전 및 가용 영역 사용

이 Node.js 코드 예제는 다음을 보여 줍니다.

- 리전과 가용 영역에 대한 설명을 검색하는 방법을 보여 줍니다.

시나리오

Amazon EC2는 전 세계의 여러 곳에서 호스팅되고 있습니다. 이 위치들은 리전과 가용 영역으로 구성됩니다. 각 리전은 개별 지리 영역입니다. 각 리전은 가용 영역이라고 알려진 격리된 위치를 여러 개 가지고 있습니다. Amazon EC2는 인스턴스와 데이터를 여러 위치에 배치할 수 있는 기능을 제공합니다.

이 예제에서는 일련의 Node.js 모듈을 사용하여 리전과 가용 영역에 대한 세부 정보를 검색합니다. Node.js 모듈은 SDK for JavaScript로 아래의 Amazon EC2 클라이언트 클래스 메서드를 사용하여 인스턴스를 관리합니다

- `describeAvailabilityZones`
- `describeRegions`

리전 및 가용 영역에 대한 자세한 내용은 Amazon EC2 사용 설명서의 리전 및 가용 영역 또는 Amazon EC2 사용 설명서의 리전 및 가용 영역을 참조하세요.

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 이러한 작업들을 완료해야 합니다.

- Node.js를 설치합니다. Node.js 설치에 대한 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 공유 자격 증명 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.

리전 및 가용 영역 설명

파일 이름이 `ec2_describeregionsandzones.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. Amazon EC2에 액세스하려면 `AWS.EC2` 서비스 객체를 생성합니다. 파라미터로 전달할 비어 있는 JSON 객체를 생성합니다. 이 객체는 사용 가능한 모든 설명을 반환합니다. 그런 다음 `describeRegions` 및 `describeAvailabilityZones` 메서드를 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });
```

```
// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

var params = {};

// Retrieves all regions/endpoints that work with EC2
ec2.describeRegions(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Regions: ", data.Regions);
  }
});

// Retrieves availability zones only for region of the ec2 service object
ec2.describeAvailabilityZones(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Availability Zones: ", data.AvailabilityZones);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node ec2_describeregionsandzones.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

Amazon EC2의 보안 그룹 작업

이 Node.js 코드 예제는 다음을 보여 줍니다.

- 보안 그룹에 대한 정보를 검색하는 방법.
- Amazon EC2 인스턴스에 액세스하기 위한 보안 그룹을 생성하는 방법
- 기존 보안 그룹을 삭제하는 방법.

시나리오

Amazon EC2 보안 그룹은 하나 이상의 인스턴스에 대한 트래픽을 제어하는 가상 방화벽 역할을 합니다. 연결된 인스턴스에서 트래픽을 주고 받을 수 있도록 각 보안 그룹에 규칙을 추가합니다. 언제든지 보안 그룹에 대한 규칙을 수정할 수 있습니다. 새 규칙은 보안 그룹과 연결된 모든 인스턴스에 자동으로 적용됩니다.

이 예제에서는 일련의 Node.js 모듈을 사용하여 보안 그룹과 관련된 여러 가지 Amazon EC2 작업을 수행합니다. Node.js 모듈은 SDK for JavaScript로 아래의 Amazon EC2 클라이언트 클래스 메서드를 사용하여 인스턴스를 관리합니다

- `describeSecurityGroups`
- `authorizeSecurityGroupIngress`
- `createSecurityGroup`
- `describeVpcs`
- `deleteSecurityGroup`

Amazon EC2 보안 그룹에 대한 자세한 내용은 Amazon EC2 사용 설명서의 Linux 인스턴스용 Amazon EC2 Amazon 보안 그룹 또는 Amazon EC2 사용 설명서의 Windows 인스턴스용 Amazon EC2 보안 그룹을 참조하세요.

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 다음 작업을 완료합니다.

- Node.js를 설치합니다. Node.js 설치에 대한 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 공유 자격 증명 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.

보안 그룹 설명

파일 이름이 `ec2_describesecuritygroups.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. Amazon EC2에 액세스하려면 `AWS.EC2` 서비스 객체를 생성합니다. 설명하려는 보안 그룹의 그룹 ID를 포함하여 파라미터로 전달할 JSON 객체를 생성합니다. 그런 다음 Amazon EC2 서비스 객체의 `describeSecurityGroups` 메서드를 직접 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
```

```
// Set the region
AWS.config.update({ region: "REGION" });

// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

var params = {
  GroupIds: ["SECURITY_GROUP_ID"],
};

// Retrieve security group descriptions
ec2.describeSecurityGroups(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", JSON.stringify(data.SecurityGroups));
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node ec2_describesecuritygroups.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

보안 그룹 및 규칙 생성

파일 이름이 `ec2_createsecuritygroup.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. Amazon EC2에 액세스하려면 `AWS.EC2` 서비스 객체를 생성합니다. 보안 그룹의 이름, 설명, VPC의 ID를 지정하는 파라미터를 위한 JSON 객체를 생성합니다. 이 파라미터를 `createSecurityGroup` 메서드에 전달합니다.

보안 그룹을 성공적으로 생성한 후 인바운드 트래픽을 허용하기 위한 규칙을 정의할 수 있습니다. Amazon EC2 인스턴스가 트래픽을 수신할 IP 프로토콜과 인바운드 포트를 지정하는 파라미터를 위한 JSON 객체를 생성합니다. 이 파라미터를 `authorizeSecurityGroupIngress` 메서드에 전달합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Load credentials and set region from JSON file
AWS.config.update({ region: "REGION" });
```

```
// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

// Variable to hold a ID of a VPC
var vpc = null;

// Retrieve the ID of a VPC
ec2.describeVpcs(function (err, data) {
  if (err) {
    console.log("Cannot retrieve a VPC", err);
  } else {
    vpc = data.Vpcs[0].VpcId;
    var paramsSecurityGroup = {
      Description: "DESCRIPTION",
      GroupName: "SECURITY_GROUP_NAME",
      VpcId: vpc,
    };
    // Create the instance
    ec2.createSecurityGroup(paramsSecurityGroup, function (err, data) {
      if (err) {
        console.log("Error", err);
      } else {
        var SecurityGroupId = data.GroupId;
        console.log("Success", SecurityGroupId);
        var paramsIngress = {
          GroupId: "SECURITY_GROUP_ID",
          IpPermissions: [
            {
              IpProtocol: "tcp",
              FromPort: 80,
              ToPort: 80,
              IpRanges: [{ CidrIp: "0.0.0.0/0" }],
            },
            {
              IpProtocol: "tcp",
              FromPort: 22,
              ToPort: 22,
              IpRanges: [{ CidrIp: "0.0.0.0/0" }],
            },
          ],
        };
        ec2.authorizeSecurityGroupIngress(paramsIngress, function (err, data) {
          if (err) {
            console.log("Error", err);
```

```
          } else {
            console.log("Ingress Successfully Set", data);
          }
        });
      }
    });
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node ec2_createsecuritygroup.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

보안 그룹 삭제

파일 이름이 `ec2_deletesecuritygroup.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. Amazon EC2에 액세스하려면 `AWS.EC2` 서비스 객체를 생성합니다. 삭제할 보안 그룹의 이름을 지정할 JSON 파라미터를 생성합니다. 그런 다음 `deleteSecurityGroup` 메서드를 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

var params = {
  GroupId: "SECURITY_GROUP_ID",
};

// Delete the security group
ec2.deleteSecurityGroup(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Security Group Deleted");
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node ec2_deletesecuritygroup.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

Amazon EC2에서 탄력적 IP 주소 사용

이 Node.js 코드 예제는 다음을 보여 줍니다.

- 탄력적 IP 주소에 대한 설명을 검색하는 방법.
- 탄력적 IP 주소를 할당하고 릴리스하는 방법.
- Amazon EC2 인스턴스와 탄력적 IP 주소를 연결하는 방법

시나리오

탄력적 IP 주소는 동적 클라우드 컴퓨팅을 위해 설계된 정적 IP 주소입니다. 탄력적 IP 주소는 AWS 계정과 연결됩니다. 퍼블릭 IP 주소로, 인터넷에서 연결할 수 있습니다. 인스턴스에 퍼블릭 IP 주소가 없는 경우 탄력적 IP 주소를 인스턴스에 연결하여 인터넷과의 통신을 활성화할 수 있습니다.

이 예제에서는 일련의 Node.js 모듈을 사용하여 탄력적 IP 주소와 관련된 여러 가지 Amazon EC2 작업을 수행합니다. Node.js 모듈은 SDK for JavaScript로 Amazon EC2 클라이언트 클래스의 다음 메서드를 사용하여 탄력적 IP 주소를 관리합니다.

- `describeAddresses`
- `allocateAddress`
- `associateAddress`
- `releaseAddress`

Amazon EC2의 탄력적 IP 주소에 대한 자세한 내용은 Amazon EC2 사용 설명서의 탄력적 IP 주소 또는 Amazon EC2 사용 설명서의 탄력적 IP 주소를 참조하세요.

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 다음 작업을 완료합니다.

- Node.js를 설치합니다. Node.js 설치에 대한 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 공유 자격 증명 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.
- Amazon EC2 인스턴스 생성 Amazon EC2 인스턴스 생성에 대한 자세한 내용은 Amazon EC2 사용 설명서의 Amazon EC2 인스턴스 또는 Amazon EC2 사용 설명서의 Amazon EC2 인스턴스를 참조하세요.

탄력적 IP 주소 설명

파일 이름이 `ec2_describeaddresses.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. Amazon EC2에 액세스하려면 `AWS.EC2` 서비스 객체를 생성합니다. VPC에 있는 항목에서 반환되는 주소를 필터링하는 파라미터로 전달할 JSON 객체를 생성합니다. 모든 탄력적 IP 주소에 대한 설명을 검색하려면 파라미터 JSON에서 필터를 생략합니다. 그런 다음 Amazon EC2 서비스 객체의 `describeAddresses` 메서드를 직접 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

var params = {
  Filters: [{ Name: "domain", Values: ["vpc"] }],
};

// Retrieve Elastic IP address descriptions
ec2.describeAddresses(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", JSON.stringify(data.Addresses));
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node ec2_describeaddresses.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

탄력적 IP 주소를 할당하고 Amazon EC2 인스턴스와 연결

파일 이름이 `ec2_allocateaddress.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. Amazon EC2에 액세스하려면 `AWS.EC2` 서비스 객체를 생성합니다. 탄력적 IP 주소를 할당하는 데 사용되는 파라미터를 위한 JSON 객체를 생성합니다. 이 경우에는 `Domain`을 VPC로 지정합니다. Amazon EC2 서비스 객체의 `allocateAddress` 메서드를 직접 호출합니다.

호출이 성공하는 경우 콜백 함수에 대한 `data` 파라미터에는 할당된 탄력적 IP 주소를 식별하는 `AllocationId` 속성이 있습니다.

새로 할당된 주소의 `AllocationId`와 Amazon EC2 인스턴스의 `InstanceId`를 포함하여 탄력적 IP 주소를 Amazon EC2 인스턴스에 연결하는 데 사용되는 파라미터를 위한 JSON 객체를 생성합니다. 그런 다음 Amazon EC2 서비스 객체의 `associateAddresses` 메서드를 직접 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

var paramsAllocateAddress = {
  Domain: "vpc",
};

// Allocate the Elastic IP address
ec2.allocateAddress(paramsAllocateAddress, function (err, data) {
  if (err) {
    console.log("Address Not Allocated", err);
  } else {
    console.log("Address allocated:", data.AllocationId);
    var paramsAssociateAddress = {
      AllocationId: data.AllocationId,
      InstanceId: "INSTANCE_ID",
```

```
    };
    // Associate the new Elastic IP address with an EC2 instance
    ec2.associateAddress(paramsAssociateAddress, function (err, data) {
      if (err) {
        console.log("Address Not Associated", err);
      } else {
        console.log("Address associated:", data.AssociationId);
      }
    });
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node ec2_allocateaddress.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

탄력적 IP 주소 해제

파일 이름이 `ec2_releaseaddress.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. Amazon EC2에 액세스하려면 `AWS.EC2` 서비스 객체를 생성합니다. 탄력적 IP 주소를 릴리스하는 데 사용되는 파라미터를 위한 JSON 객체를 생성합니다. 이 경우에는 탄력적 IP 주소의 `AllocationId`가 지정됩니다. 탄력적 IP 주소를 릴리스하면 Amazon EC2 인스턴스에서 연결이 해제됩니다. Amazon EC2 서비스 객체의 `releaseAddress` 메서드를 직접 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

var paramsReleaseAddress = {
  AllocationId: "ALLOCATION_ID",
};

// Disassociate the Elastic IP address from EC2 instance
ec2.releaseAddress(paramsReleaseAddress, function (err, data) {
  if (err) {
    console.log("Error", err);
```

```
  } else {
    console.log("Address released");
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node ec2_releaseaddress.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

AWS Elemental MediaConvert 예제

AWS Elemental MediaConvert는 브로드캐스트급 기능을 갖춘 파일 기반의 비디오 트랜스코딩 서비스입니다. 이 서비스를 사용하여 인터넷에서 브로드캐스트 및 비디오 온디맨드(VOD) 전달용 자산을 생성할 수 있습니다. 자세한 설명은 AWS Elemental MediaConvert사용자 가이드를 참조하세요.

MediaConvert용 JavaScript API는 `AWS.MediaConvert` 클라이언트 클래스를 통해 노출됩니다. 자세한 내용은 API 참조의 `Class: AWS.MediaConvert`를 참조하세요.

주제

- MediaConvert에서 트랜스코딩 작업 생성 및 관리
- MediaConvert에서 작업 템플릿 사용

MediaConvert에서 트랜스코딩 작업 생성 및 관리

이 Node.js 코드 예제는 다음을 보여 줍니다.

- MediaConvert에서 트랜스코딩 작업을 생성하는 방법
- 트랜스코딩 작업을 취소하는 방법.
- 완료된 트랜스코딩 작업에 대한 JSON을 검색하는 방법.
- 최근에 생성된 최대 20개 작업에 대한 JSON 배열을 검색하는 방법.

시나리오

이 예에서는 Node.js 모듈을 사용하여 트랜스코딩 작업을 생성하고 관리할 MediaConvert를 직접적으로 호출합니다. 이 코드는 SDK for JavaScript에서 MediaConvert 클라이언트 클래스의 다음 메서드를 사용하여 이 작업을 수행합니다.

- `createJob`
- `cancelJob`
- `getJob`
- `listJobs`

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 다음 작업을 완료합니다.

- Node.js를 설치합니다. 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 공유 자격 증명 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.
- 작업 입력 파일 및 출력 파일을 위한 스토리지를 제공하는 Amazon S3 버킷을 생성하고 구성합니다. 자세한 내용은 AWS Elemental MediaConvert 사용 설명서의 파일 스토리지 생성 섹션을 참조하세요.
- 입력 스토리지를 위해 프로비저닝한 Amazon S3 버킷에 입력 비디오를 업로드합니다. 지원되는 입력 비디오 코덱 및 컨테이너 목록은 AWS Elemental MediaConvert 사용 설명서의 지원되는 입력 코덱 및 컨테이너를 참조하세요.
- 출력 파일이 저장된 Amazon S3 버킷 및 입력 파일에 대한 액세스 권한을 MediaConvert에 부여하는 IAM 역할을 생성합니다. 자세한 내용은 AWS Elemental MediaConvert 사용 설명서의 IAM 권한 설정하기를 참조하세요.

단순 트랜스코딩 작업 정의

파일 이름이 `emc_createjob.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. 트랜스코드 작업 파라미터를 정의하는 JSON을 생성합니다.

이러한 파라미터는 매우 세부적입니다. AWS Elemental MediaConvert 콘솔을 사용하면 콘솔에서 작업 설정을 선택한 다음, 작업 섹션 하단에서 작업 JSON 표시를 선택하여 JSON 작업 파라미터를 생성할 수 있습니다. 이 예제는 단순 작업용 JSON을 보여줍니다.

```
var params = {
  Queue: "JOB_QUEUE_ARN",
  UserMetadata: {
    Customer: "Amazon",
  },
  Role: "IAM_ROLE_ARN",
  Settings: {
    OutputGroups: [
      {
        Name: "File Group",
        OutputGroupSettings: {
          Type: "FILE_GROUP_SETTINGS",
          FileGroupSettings: {
            Destination: "s3://OUTPUT_BUCKET_NAME/",
          },
        },
        Outputs: [
          {
            VideoDescription: {
              ScalingBehavior: "DEFAULT",
              TimecodeInsertion: "DISABLED",
              AntiAlias: "ENABLED",
              Sharpness: 50,
              CodecSettings: {
                Codec: "H_264",
                H264Settings: {
                  InterlaceMode: "PROGRESSIVE",
                  NumberReferenceFrames: 3,
                  Syntax: "DEFAULT",
                  Softness: 0,
                  GopClosedCadence: 1,
                  GopSize: 90,
                  Slices: 1,
                  GopBReference: "DISABLED",
                  SlowPal: "DISABLED",
                  SpatialAdaptiveQuantization: "ENABLED",
                  TemporalAdaptiveQuantization: "ENABLED",
                  FlickerAdaptiveQuantization: "DISABLED",
                  EntropyEncoding: "CABAC",
                  Bitrate: 5000000,
                  FramerateControl: "SPECIFIED",
                  RateControlMode: "CBR",
                  CodecProfile: "MAIN",
```

```
              Telecine: "NONE",
              MinIInterval: 0,
              AdaptiveQuantization: "HIGH",
              CodecLevel: "AUTO",
              FieldEncoding: "PAFF",
              SceneChangeDetect: "ENABLED",
              QualityTuningLevel: "SINGLE_PASS",
              FramerateConversionAlgorithm: "DUPLICATE_DROP",
              UnregisteredSeiTimecode: "DISABLED",
              GopSizeUnits: "FRAMES",
              ParControl: "SPECIFIED",
              NumberBFramesBetweenReferenceFrames: 2,
              RepeatPps: "DISABLED",
              FramerateNumerator: 30,
              FramerateDenominator: 1,
              ParNumerator: 1,
              ParDenominator: 1,
            },
          },
          AfdSignaling: "NONE",
          DropFrameTimecode: "ENABLED",
          RespondToAfd: "NONE",
          ColorMetadata: "INSERT",
        },
        AudioDescriptions: [
          {
            AudioTypeControl: "FOLLOW_INPUT",
            CodecSettings: {
              Codec: "AAC",
              AacSettings: {
                AudioDescriptionBroadcasterMix: "NORMAL",
                RateControlMode: "CBR",
                CodecProfile: "LC",
                CodingMode: "CODING_MODE_2_0",
                RawFormat: "NONE",
                SampleRate: 48000,
                Specification: "MPEG4",
                Bitrate: 64000,
              },
            },
            LanguageCodeControl: "FOLLOW_INPUT",
            AudioSourceName: "Audio Selector 1",
          },
        ],
```

```
            ContainerSettings: {
              Container: "MP4",
              Mp4Settings: {
                CslgAtom: "INCLUDE",
                FreeSpaceBox: "EXCLUDE",
                MoovPlacement: "PROGRESSIVE_DOWNLOAD",
              },
            },
            NameModifier: "_1",
          },
        ],
      },
    ],
    AdAvailOffset: 0,
    Inputs: [
      {
        AudioSelectors: {
          "Audio Selector 1": {
            Offset: 0,
            DefaultSelection: "NOT_DEFAULT",
            ProgramSelection: 1,
            SelectorType: "TRACK",
            Tracks: [1],
          },
        },
        VideoSelector: {
          ColorSpace: "FOLLOW",
        },
        FilterEnable: "AUTO",
        PsiControl: "USE_PSI",
        FilterStrength: 0,
        DeblockFilter: "DISABLED",
        DenoiseFilter: "DISABLED",
        TimecodeSource: "EMBEDDED",
        FileInput: "s3://INPUT_BUCKET_AND_FILE_NAME",
      },
    ],
    TimecodeConfig: {
      Source: "EMBEDDED",
    },
  },
};
```

트랜스코딩 작업 생성

작업 파라미터 JSON을 생성한 후, 파라미터를 전달하는 `createJob` 서비스 객체를 호출하기 위한 promise를 생성하여 `AWS.MediaConvert` 메서드를 호출합니다. 그런 다음 promise 콜백에서 response를 처리합니다. 생성된 작업의 ID는 응답 `data`에서 반환됩니다.

```
// Create a promise on a MediaConvert object
var endpointPromise = new AWS.MediaConvert({ apiVersion: "2017-08-29" })
  .createJob(params)
  .promise();

// Handle promise's fulfilled/rejected status
endpointPromise.then(
  function (data) {
    console.log("Job created! ", data);
  },
  function (err) {
    console.log("Error", err);
  }
);
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node emc_createjob.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

트랜스코딩 작업 취소

파일 이름이 `emc_canceljob.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. 취소할 작업의 ID가 포함된 JSON을 생성합니다. 그런 다음 파라미터를 전달하는 `cancelJob` 서비스 객체를 호출하기 위한 promise를 생성하여 `AWS.MediaConvert` 메서드를 호출합니다. promise 콜백에서 응답을 처리합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the Region
AWS.config.update({ region: "us-west-2" });
// Set MediaConvert to customer endpoint
AWS.config.mediaconvert = { endpoint: "ACCOUNT_ENDPOINT" };
```

```
var params = {
  Id: "JOB_ID" /* required */,
};

// Create a promise on a MediaConvert object
var endpointPromise = new AWS.MediaConvert({ apiVersion: "2017-08-29" })
  .cancelJob(params)
  .promise();

// Handle promise's fulfilled/rejected status
endpointPromise.then(
  function (data) {
    console.log("Job  " + params.Id + " is canceled");
  },
  function (err) {
    console.log("Error", err);
  }
);
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node ec2_canceljob.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

최근 트랜스코딩 작업 나열

파일 이름이 `emc_listjobs.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다.

목록을 `ASCENDING` 또는 `DESCENDING` 순서로 정렬할지 여부, 확인할 작업 대기열의 ARN, 포함할 작업의 상태 등을 지정하는 값을 포함하여 파라미터 JSON을 생성합니다. 그런 다음 파라미터를 전달하는 `listJobs` 서비스 객체를 호출하기 위한 promise를 생성하여 `AWS.MediaConvert` 메서드를 호출합니다. promise 콜백에서 응답을 처리합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the Region
AWS.config.update({ region: "us-west-2" });
// Set the customer endpoint
AWS.config.mediaconvert = { endpoint: "ACCOUNT_ENDPOINT" };

var params = {
```

```
  MaxResults: 10,
  Order: "ASCENDING",
  Queue: "QUEUE_ARN",
  Status: "SUBMITTED",
};

// Create a promise on a MediaConvert object
var endpointPromise = new AWS.MediaConvert({ apiVersion: "2017-08-29" })
  .listJobs(params)
  .promise();

// Handle promise's fulfilled/rejected status
endpointPromise.then(
  function (data) {
    console.log("Jobs: ", data);
  },
  function (err) {
    console.log("Error", err);
  }
);
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node emc_listjobs.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

MediaConvert에서 작업 템플릿 사용

이 Node.js 코드 예제는 다음을 보여 줍니다.

- MediaConvert 작업 템플릿을 생성하는 방법
- 작업 템플릿을 사용하여 트랜스코딩 작업을 생성하는 방법.
- 모든 작업 템플릿의 목록을 표시하는 방법.
- 작업 템플릿을 삭제하는 방법.

시나리오

MediaConvert에서 트랜스코딩 작업을 생성하는 데 필요한 JSON은 많은 수의 설정을 포함하여 세부적입니다. 후속 작업을 생성하는 데 사용할 수 있는 작업 템플릿에 알려진 좋은 설정을 저장하여 작업 생성을 대폭 간소화할 수 있습니다. 이 예에서는 Node.js 모듈을 사용해 MediaConvert를 직접적으로 호출하여 작업 템플릿을 생성, 사용 및 관리합니다. 이 코드는 SDK for JavaScript에서 MediaConvert 클라이언트 클래스의 다음 메서드를 사용하여 이 작업을 수행합니다.

- `createJobTemplate`
- `createJob`
- `deleteJobTemplate`
- `listJobTemplates`

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 다음 작업을 완료합니다.

- Node.js를 설치합니다. 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 공유 자격 증명 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.
- 출력 파일이 저장된 Amazon S3 버킷 및 입력 파일에 대한 액세스 권한을 MediaConvert에 부여하는 IAM 역할을 생성합니다. 자세한 내용은 AWS Elemental MediaConvert 사용 설명서의 IAM 권한 설정하기를 참조하세요.

작업 템플릿 생성

파일 이름이 `emc_create_jobtemplate.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다.

템플릿 생성을 위한 파라미터 JSON을 지정합니다. 이전에 성공한 작업에서 대부분의 JSON 파라미터를 사용하여 템플릿에서 `Settings` 값을 지정할 수 있습니다. 이 예제에서는 MediaConvert에서 트랜스코딩 작업 생성 및 관리의 작업 설정을 사용합니다.

파라미터를 전달하는 `createJobTemplate` 서비스 객체를 호출하기 위한 promise를 생성하여 `AWS.MediaConvert` 메서드를 호출합니다. 그런 다음 promise 콜백에서 response를 처리합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
```

```
// Set the Region
AWS.config.update({ region: "us-west-2" });
// Set the custom endpoint for your account
AWS.config.mediaconvert = { endpoint: "ACCOUNT_ENDPOINT" };

var params = {
  Category: "YouTube Jobs",
  Description: "Final production transcode",
  Name: "DemoTemplate",
  Queue: "JOB_QUEUE_ARN",
  Settings: {
    OutputGroups: [
      {
        Name: "File Group",
        OutputGroupSettings: {
          Type: "FILE_GROUP_SETTINGS",
          FileGroupSettings: {
            Destination: "s3://BUCKET_NAME/",
          },
        },
        Outputs: [
          {
            VideoDescription: {
              ScalingBehavior: "DEFAULT",
              TimecodeInsertion: "DISABLED",
              AntiAlias: "ENABLED",
              Sharpness: 50,
              CodecSettings: {
                Codec: "H_264",
                H264Settings: {
                  InterlaceMode: "PROGRESSIVE",
                  NumberReferenceFrames: 3,
                  Syntax: "DEFAULT",
                  Softness: 0,
                  GopClosedCadence: 1,
                  GopSize: 90,
                  Slices: 1,
                  GopBReference: "DISABLED",
                  SlowPal: "DISABLED",
                  SpatialAdaptiveQuantization: "ENABLED",
                  TemporalAdaptiveQuantization: "ENABLED",
                  FlickerAdaptiveQuantization: "DISABLED",
                  EntropyEncoding: "CABAC",
                  Bitrate: 5000000,
```

```
              FramerateControl: "SPECIFIED",
              RateControlMode: "CBR",
              CodecProfile: "MAIN",
              Telecine: "NONE",
              MinIInterval: 0,
              AdaptiveQuantization: "HIGH",
              CodecLevel: "AUTO",
              FieldEncoding: "PAFF",
              SceneChangeDetect: "ENABLED",
              QualityTuningLevel: "SINGLE_PASS",
              FramerateConversionAlgorithm: "DUPLICATE_DROP",
              UnregisteredSeiTimecode: "DISABLED",
              GopSizeUnits: "FRAMES",
              ParControl: "SPECIFIED",
              NumberBFramesBetweenReferenceFrames: 2,
              RepeatPps: "DISABLED",
              FramerateNumerator: 30,
              FramerateDenominator: 1,
              ParNumerator: 1,
              ParDenominator: 1,
            },
          },
          AfdSignaling: "NONE",
          DropFrameTimecode: "ENABLED",
          RespondToAfd: "NONE",
          ColorMetadata: "INSERT",
        },
        AudioDescriptions: [
          {
            AudioTypeControl: "FOLLOW_INPUT",
            CodecSettings: {
              Codec: "AAC",
              AacSettings: {
                AudioDescriptionBroadcasterMix: "NORMAL",
                RateControlMode: "CBR",
                CodecProfile: "LC",
                CodingMode: "CODING_MODE_2_0",
                RawFormat: "NONE",
                SampleRate: 48000,
                Specification: "MPEG4",
                Bitrate: 64000,
              },
            },
            LanguageCodeControl: "FOLLOW_INPUT",
```

```
                AudioSourceName: "Audio Selector 1",
              },
            ],
            ContainerSettings: {
              Container: "MP4",
              Mp4Settings: {
                CslgAtom: "INCLUDE",
                FreeSpaceBox: "EXCLUDE",
                MoovPlacement: "PROGRESSIVE_DOWNLOAD",
              },
            },
            NameModifier: "_1",
          },
        ],
      },
    ],
    AdAvailOffset: 0,
    Inputs: [
      {
        AudioSelectors: {
          "Audio Selector 1": {
            Offset: 0,
            DefaultSelection: "NOT_DEFAULT",
            ProgramSelection: 1,
            SelectorType: "TRACK",
            Tracks: [1],
          },
        },
        VideoSelector: {
          ColorSpace: "FOLLOW",
        },
        FilterEnable: "AUTO",
        PsiControl: "USE_PSI",
        FilterStrength: 0,
        DeblockFilter: "DISABLED",
        DenoiseFilter: "DISABLED",
        TimecodeSource: "EMBEDDED",
      },
    ],
    TimecodeConfig: {
      Source: "EMBEDDED",
    },
  },
};
```

```
// Create a promise on a MediaConvert object
var templatePromise = new AWS.MediaConvert({ apiVersion: "2017-08-29" })
  .createJobTemplate(params)
  .promise();

// Handle promise's fulfilled/rejected status
templatePromise.then(
  function (data) {
    console.log("Success!", data);
  },
  function (err) {
    console.log("Error", err);
  }
);
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node emc_create_jobtemplate.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

작업 템플릿에서 트랜스코딩 작업 생성

파일 이름이 `emc_template_createjob.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다.

사용할 작업 템플릿의 이름, 생성하는 작업에 특정하게 사용할 `Settings`를 포함하여 작업 생성 파라미터 JSON을 생성합니다. 그런 다음 파라미터를 전달하는 `createJobs` 서비스 객체를 호출하기 위한 promise를 생성하여 `AWS.MediaConvert` 메서드를 호출합니다. promise 콜백에서 응답을 처리합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the Region
AWS.config.update({ region: "us-west-2" });
// Set the custom endpoint for your account
AWS.config.mediaconvert = { endpoint: "ACCOUNT_ENDPOINT" };

var params = {
  Queue: "QUEUE_ARN",
  JobTemplate: "TEMPLATE_NAME",
```

```
  Role: "ROLE_ARN",
  Settings: {
    Inputs: [
      {
        AudioSelectors: {
          "Audio Selector 1": {
            Offset: 0,
            DefaultSelection: "NOT_DEFAULT",
            ProgramSelection: 1,
            SelectorType: "TRACK",
            Tracks: [1],
          },
        },
        VideoSelector: {
          ColorSpace: "FOLLOW",
        },
        FilterEnable: "AUTO",
        PsiControl: "USE_PSI",
        FilterStrength: 0,
        DeblockFilter: "DISABLED",
        DenoiseFilter: "DISABLED",
        TimecodeSource: "EMBEDDED",
        FileInput: "s3://BUCKET_NAME/FILE_NAME",
      },
    ],
  },
};

// Create a promise on a MediaConvert object
var templateJobPromise = new AWS.MediaConvert({ apiVersion: "2017-08-29" })
  .createJob(params)
  .promise();

// Handle promise's fulfilled/rejected status
templateJobPromise.then(
  function (data) {
    console.log("Success! ", data);
  },
  function (err) {
    console.log("Error", err);
  }
);
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node emc_template_createjob.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

작업 템플릿 목록 표시

파일 이름이 emc_listtemplates.js인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다.

listTemplates 클라이언트 클래스의 AWS.MediaConvert 메서드에 대한 요청 파라미터를 전달할 객체를 생성합니다. 나열할 템플릿(NAME, CREATION DATE, SYSTEM), 나열할 개수 및 정렬 순서를 결정하는 값을 포함시킵니다. listTemplates 메서드를 직접 호출하려면 MediaConvert 서비스 객체를 간접 호출하기 위한 promise를 생성하고 파라미터를 전달합니다. 그런 다음 promise 콜백에서 response를 처리합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the Region
AWS.config.update({ region: "us-west-2" });
// Set the customer endpoint
AWS.config.mediaconvert = { endpoint: "ACCOUNT_ENDPOINT" };

var params = {
  ListBy: "NAME",
  MaxResults: 10,
  Order: "ASCENDING",
};

// Create a promise on a MediaConvert object
var listTemplatesPromise = new AWS.MediaConvert({ apiVersion: "2017-08-29" })
  .listJobTemplates(params)
  .promise();

// Handle promise's fulfilled/rejected status
listTemplatesPromise.then(
  function (data) {
    console.log("Success ", data);
  },
  function (err) {
    console.log("Error", err);
```

```
  }
);
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node emc_listtemplates.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

작업 템플릿 삭제

파일 이름이 `emc_deletetemplate.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다.

삭제하려는 작업 템플릿의 이름을 `deleteJobTemplate` 클라이언트 클래스의 `AWS.MediaConvert` 메서드에 대한 파라미터로 전달할 객체를 생성합니다. `deleteJobTemplate` 메서드를 직접 호출하려면 MediaConvert 서비스 객체를 간접 호출하기 위한 promise를 생성하고 파라미터를 전달합니다. promise 콜백에서 응답을 처리합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the Region
AWS.config.update({ region: "us-west-2" });
// Set the customer endpoint
AWS.config.mediaconvert = { endpoint: "ACCOUNT_ENDPOINT" };

var params = {
  Name: "TEMPLATE_NAME",
};

// Create a promise on a MediaConvert object
var deleteTemplatePromise = new AWS.MediaConvert({ apiVersion: "2017-08-29" })
  .deleteJobTemplate(params)
  .promise();

// Handle promise's fulfilled/rejected status
deleteTemplatePromise.then(
  function (data) {
    console.log("Success ", data);
  },
  function (err) {
```

```
    console.log("Error", err);
  }
);
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node emc_deletetemplate.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

AWS IAM 예제

AWS Identity and Access Management(IAM)는 Amazon Web Services 고객이 AWS에서 사용자와 각 사용자 권한을 관리할 수 있도록 하는 웹 서비스입니다. 이 서비스는 클라우드 내에 AWS 제품을 사용하는 여러 사용자 또는 시스템이 있는 조직을 대상으로 합니다. IAM을 사용하면 사용자, 액세스 키와 같은 보안 자격 증명, 사용자가 액세스할 수 있는 AWS 리소스를 제어하는 권한을 중앙 관리할 수 있습니다.

IAM용 JavaScript API는 `AWS.IAM` 클라이언트 클래스를 통해 노출됩니다. IAM 클라이언트 클래스 사용에 대한 자세한 내용은 API 참조의 `Class: AWS.IAM` 섹션을 참조하세요.

주제

- IAM 사용자 관리
- IAM 정책 작업
- IAM 액세스 키 관리
- IAM 서버 인증서 작업
- IAM 계정 별칭 관리

IAM 사용자 관리

이 Node.js 코드 예제는 다음을 보여 줍니다.

- IAM 사용자 목록을 검색하는 방법
- 사용자를 생성하고 삭제하는 방법.
- 사용자 이름을 업데이트하는 방법.

시나리오

이 예제에서는 일련의 Node.js 모듈을 사용하여 IAM에서 사용자를 생성하고 관리합니다. Node.js 모듈은 SDK for JavaScript로 `AWS.IAM` 클라이언트 클래스의 다음 메서드를 사용하여 사용자를 생성, 삭제 및 업데이트합니다.

- `createUser`
- `listUsers`
- `updateUser`
- `getUser`
- `deleteUser`

IAM 사용자에 대한 자세한 내용은 IAM 사용 설명서의 IAM 사용자 섹션을 참조하세요.

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 이러한 작업들을 완료해야 합니다.

- Node.js를 설치합니다. Node.js 설치에 대한 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 공유 자격 증명 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.

사용자 생성

파일 이름이 `iam_createuser.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. IAM에 액세스하려면 `AWS.IAM` 서비스 객체를 생성합니다. 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. 이 객체에는 새 사용자에 사용하려는 사용자 이름이 명령줄 파라미터로 포함됩니다.

사용자 이름이 이미 있는 경우 `getUser` 서비스 객체의 `AWS.IAM`메서드를 호출합니다. 사용자 이름이 현재 없는 경우 `createUser` 메서드를 호출하여 사용자 이름을 생성합니다. 이름이 이미 있는 경우에는 그런 취지의 메시지를 콘솔에 표시합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

var params = {
  UserName: process.argv[2],
};

iam.getUser(params, function (err, data) {
  if (err && err.code === "NoSuchEntity") {
    iam.createUser(params, function (err, data) {
      if (err) {
        console.log("Error", err);
      } else {
        console.log("Success", data);
      }
    });
  } else {
    console.log(
      "User " + process.argv[2] + " already exists",
      data.User.UserId
    );
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node iam_createuser.js USER_NAME
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

계정의 사용자 목록 표시

파일 이름이 `iam_listusers.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. IAM에 액세스하려면 `AWS.IAM` 서비스 객체를 생성합니다. 사용자 목록을 표시하는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. 이 예제에서는 `MaxItems` 파라미터를 10으로 설정하여 반환되는 수를 제한합니다. `listUsers` 서비스 객체의 `AWS.IAM` 메서드를 호출합니다. 첫 번째 사용자의 이름과 생성 날짜를 콘솔에 씁니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

var params = {
  MaxItems: 10,
};

iam.listUsers(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    var users = data.Users || [];
    users.forEach(function (user) {
      console.log("User " + user.UserName + " created", user.CreateDate);
    });
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node iam_listusers.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

사용자 이름 업데이트

파일 이름이 `iam_updateuser.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. IAM에 액세스하려면 `AWS.IAM` 서비스 객체를 생성합니다. 사용자 목록을 표시하는 데 필요한 파라미터를 포함하는 JSON 객체를 생성하고, 현재 및 새 사용자 이름을 모두 명령줄 파라미터로 지정합니다. `updateUser` 서비스 객체의 `AWS.IAM` 메서드를 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

var params = {
  UserName: process.argv[2],
  NewUserName: process.argv[3],
};

iam.updateUser(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

예제를 실행하려면 사용자의 현재 이름을 지정한 다음 새 사용자 이름을 지정하여 명령줄에 다음을 입력합니다.

```
node iam_updateuser.js ORIGINAL_USERNAME NEW_USERNAME
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

사용자 삭제

파일 이름이 `iam_deleteuser.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. IAM에 액세스하려면 `AWS.IAM` 서비스 객체를 생성합니다. 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. 이 객체에는 삭제하려는 사용자 이름이 명령줄 파라미터로 포함됩니다.

사용자 이름이 이미 있는 경우 getUser 서비스 객체의 AWS.IAM메서드를 호출합니다. 사용자 이름이 없는 경우 이를 알리는 메시지를 콘솔에 씁니다. 사용자가 있는 경우 deleteUser 메서드를 호출하여 사용자를 삭제합니다.

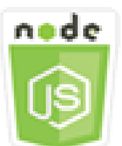
```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

var params = {
  UserName: process.argv[2],
};

iam.getUser(params, function (err, data) {
  if (err && err.code === "NoSuchEntity") {
    console.log("User " + process.argv[2] + " does not exist.");
  } else {
    iam.deleteUser(params, function (err, data) {
      if (err) {
        console.log("Error", err);
      } else {
        console.log("Success", data);
      }
    });
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node iam_deleteuser.js USER_NAME
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

IAM 정책 작업



이 Node.js 코드 예제는 다음을 보여 줍니다.

- IAM 정책을 생성하고 삭제하는 방법
- IAM 정책을 역할에 연결하고 분리하는 방법

시나리오

정책을 생성하여 사용자에게 권한을 부여합니다. 정책은 사용자가 수행할 수 있는 작업과 해당 작업이 영향을 미칠 수 있는 리소스 목록을 표시하는 문서입니다. 명시적으로 허용되지 않은 작업 또는 리소스는 기본적으로 모두 거부됩니다. 정책을 생성하여 사용자, 사용자 그룹, 사용자가 맡는 역할, 리소스에 연결할 수 있습니다.

이 예제에서는 일련의 Node.js 모듈을 사용하여 IAM에서 정책을 관리합니다. Node.js 모듈은 SDK for JavaScript로 `AWS.IAM` 클라이언트 클래스의 다음 메서드를 사용하여 정책을 생성하고 삭제하며 역할 정책을 연결하고 분리합니다.

- `createPolicy`
- `getPolicy`
- `listAttachedRolePolicies`
- `attachRolePolicy`
- `detachRolePolicy`

IAM 사용자에 대한 자세한 내용은 IAM 사용 설명서에서 액세스 관리 개요: 권한 및 정책을 참조하세요.

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 이러한 작업들을 완료해야 합니다.

- Node.js를 설치합니다. Node.js 설치에 대한 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 공유 자격 증명 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.
- 정책을 연결할 수 있는 IAM 역할을 생성합니다. IAM 역할 생성에 대한 자세한 내용은 IAM 사용 설명서의 IAM 역할 생성을 참조하세요.

IAM 정책 생성

파일 이름이 `iam_createpolicy.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. IAM에 액세스하려면 `AWS.IAM` 서비스 객체를 생성합니다. 두 개의 JSON 객체를 생성합니다. 하나는 생성하려는 정책 문서를 포함하고 다른 하나는 정책 JSON과 정책에 지정할 이름을 포함하여 정책을 생성하는 데 필요한 파라미터를 포함합니다. 파라미터에서 정책 JSON 객체를 문자열로 만들어야 합니다. `createPolicy` 서비스 객체의 `AWS.IAM` 메서드를 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

var myManagedPolicy = {
  Version: "2012-10-17",
  Statement: [
    {
      Effect: "Allow",
      Action: "logs:CreateLogGroup",
      Resource: "RESOURCE_ARN",
    },
    {
      Effect: "Allow",
      Action: [
        "dynamodb:DeleteItem",
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:Scan",
        "dynamodb:UpdateItem",
      ],
      Resource: "RESOURCE_ARN",
    },
  ],
};

var params = {
  PolicyDocument: JSON.stringify(myManagedPolicy),
  PolicyName: "myDynamoDBPolicy",
};
```

```
iam.createPolicy(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node iam_createpolicy.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

IAM 정책 가져오기

파일 이름이 `iam_getpolicy.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. IAM에 액세스하려면 `AWS.IAM` 서비스 객체를 생성합니다. 정책을 검색하는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. 이 객체에는 가져오려는 정책의 ARN이 포함됩니다. `getPolicy` 서비스 객체의 `AWS.IAM` 메서드를 호출합니다. 정책 설명을 콘솔에 씁니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

var params = {
  PolicyArn: "arn:aws:iam::aws:policy/AWSLambdaExecute",
};

iam.getPolicy(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.Policy.Description);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node iam_getpolicy.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

관리형 역할 정책 연결

파일 이름이 `iam_attachrolepolicy.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. IAM에 액세스하려면 `AWS.IAM` 서비스 객체를 생성합니다. 역할에 연결된 관리형 IAM 정책의 목록을 가져오는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. 이 객체는 역할의 이름으로 구성됩니다. 역할 이름을 명령줄 파라미터로 제공합니다. `listAttachedRolePolicies` 서비스 객체의 `AWS.IAM` 메서드를 호출합니다. 그러면 관리형 정책 배열이 콜백 함수에 반환됩니다.

배열 구성원을 점검하여 역할에 연결하려는 정책이 이미 연결되어 있는지 확인합니다. 정책이 연결되어 있지 않은 경우 `attachRolePolicy` 메서드를 호출하여 정책을 연결합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

var paramsRoleList = {
  RoleName: process.argv[2],
};

iam.listAttachedRolePolicies(paramsRoleList, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    var myRolePolicies = data.AttachedPolicies;
    myRolePolicies.forEach(function (val, index, array) {
      if (myRolePolicies[index].PolicyName === "AmazonDynamoDBFullAccess") {
        console.log(
          "AmazonDynamoDBFullAccess is already attached to this role."
        );
        process.exit();
      }
    });
    var params = {
      PolicyArn: "arn:aws:iam::aws:policy/AmazonDynamoDBFullAccess",
```

```
      RoleName: process.argv[2],
    };
    iam.attachRolePolicy(params, function (err, data) {
      if (err) {
        console.log("Unable to attach policy to role", err);
      } else {
        console.log("Role attached successfully");
      }
    });
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node iam_attachrolepolicy.js IAM_ROLE_NAME
```

관리형 역할 정책 분리

파일 이름이 `iam_detachrolepolicy.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. IAM에 액세스하려면 `AWS.IAM` 서비스 객체를 생성합니다. 역할에 연결된 관리형 IAM 정책의 목록을 가져오는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. 이 객체는 역할의 이름으로 구성됩니다. 역할 이름을 명령줄 파라미터로 제공합니다. `listAttachedRolePolicies` 서비스 객체의 `AWS.IAM` 메서드를 호출합니다. 그러면 관리형 정책 배열이 콜백 함수에서 반환됩니다.

배열 구성원을 점검하여 역할에서 분리하려는 정책이 연결되어 있는지 확인합니다. 정책이 연결되어 있는 경우 `detachRolePolicy` 메서드를 호출하여 정책을 분리합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

var paramsRoleList = {
  RoleName: process.argv[2],
};

iam.listAttachedRolePolicies(paramsRoleList, function (err, data) {
  if (err) {
```

```
    console.log("Error", err);
  } else {
    var myRolePolicies = data.AttachedPolicies;
    myRolePolicies.forEach(function (val, index, array) {
      if (myRolePolicies[index].PolicyName === "AmazonDynamoDBFullAccess") {
        var params = {
          PolicyArn: "arn:aws:iam::aws:policy/AmazonDynamoDBFullAccess",
          RoleName: process.argv[2],
        };
        iam.detachRolePolicy(params, function (err, data) {
          if (err) {
            console.log("Unable to detach policy from role", err);
          } else {
            console.log("Policy detached from role successfully");
            process.exit();
          }
        });
      }
    });
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node iam_detachrolepolicy.js IAM_ROLE_NAME
```

IAM 액세스 키 관리

이 Node.js 코드 예제는 다음을 보여 줍니다.

- 사용자의 액세스 키를 관리하는 방법.

시나리오

사용자가 SDK for JavaScript에서 AWS를 프로그래밍 방식으로 호출하려면 사용자 고유의 액세스 키가 필요합니다. 이 요구를 충족하기 위해 IAM 사용자에 대한 액세스 키(액세스 키 ID 및 보안 액세스

키)를 생성, 수정, 확인 또는 교체할 수 있습니다. 기본적으로 액세스 키를 생성할 때 액세스 키의 상태는 `Active`이므로 사용자는 액세스 키를 API 호출에 사용할 수 있습니다.

이 예제에서는 일련의 Node.js 모듈을 사용하여 IAM에서 액세스 키를 관리합니다. Node.js 모듈은 SDK for JavaScript로 `AWS.IAM` 클라이언트 클래스의 다음 메서드를 사용하여 IAM 액세스 키를 관리합니다.

- createAccessKey
- listAccessKeys
- getAccessKeyLastUsed
- updateAccessKey
- deleteAccessKey

IAM 액세스 키에 대한 자세한 내용은 IAM 사용 설명서의 액세스 키를 참조하세요.

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 이러한 작업들을 완료해야 합니다.

- Node.js를 설치합니다. Node.js 설치에 대한 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 공유 자격 증명 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.

사용자를 위한 액세스 키 생성

파일 이름이 `iam_createaccesskeys.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. IAM에 액세스하려면 `AWS.IAM` 서비스 객체를 생성합니다. 새 액세스 키를 생성하는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. 이 객체에는 IAM 사용자의 이름이 포함됩니다. `createAccessKey` 서비스 객체의 `AWS.IAM` 메서드를 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });
```

```
iam.createAccessKey({ UserName: "IAM_USER_NAME" }, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.AccessKey);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다. 한 번만 제공할 수 있는 보안 키를 잃어버리지 않도록 반환된 데이터를 텍스트 파일에 파이프해야 합니다.

```
node iam_createaccesskeys.js > newuserkeys.txt
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

사용자의 액세스 키 목록 표시

파일 이름이 `iam_listaccesskeys.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. IAM에 액세스하려면 `AWS.IAM` 서비스 객체를 생성합니다. 사용자의 액세스 키를 검색하는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. 이 객체에는 IAM 사용자의 이름과 목록에 표시할 액세스 키 페어의 최대 수(선택 사항)가 포함됩니다. `listAccessKeys` 서비스 객체의 `AWS.IAM` 메서드를 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

var params = {
  MaxItems: 5,
  UserName: "IAM_USER_NAME",
};

iam.listAccessKeys(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
```

```
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node iam_listaccesskeys.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

액세스 키의 최종 사용 가져오기

파일 이름이 `iam_accesskeylastused.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. IAM에 액세스하려면 `AWS.IAM` 서비스 객체를 생성합니다. 새 액세스 키를 생성하는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. 이 객체에는 최종 사용 정보가 필요한 액세스 키 ID가 포함됩니다. `getAccessKeyLastUsed` 서비스 객체의 `AWS.IAM` 메서드를 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

iam.getAccessKeyLastUsed(
  { AccessKeyId: "ACCESS_KEY_ID" },
  function (err, data) {
    if (err) {
      console.log("Error", err);
    } else {
      console.log("Success", data.AccessKeyLastUsed);
    }
  }
);
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node iam_accesskeylastused.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

액세스 키 상태 업데이트

파일 이름이 iam_updateaccesskey.js인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. IAM에 액세스하려면 AWS.IAM 서비스 객체를 생성합니다. 액세스 키의 상태를 업데이트하는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. 이 객체에는 액세스 키 ID와 업데이트된 상태가 포함됩니다. 상태는 Active 또는 Inactive일 수 있습니다. updateAccessKey 서비스 객체의 AWS.IAM 메서드를 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

var params = {
  AccessKeyId: "ACCESS_KEY_ID",
  Status: "Active",
  UserName: "USER_NAME",
};

iam.updateAccessKey(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node iam_updateaccesskey.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

액세스 키 삭제

파일 이름이 iam_deleteaccesskey.js인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. IAM에 액세스하려면 AWS.IAM 서비스 객체를 생성합니다. 액세스 키를 삭제하는 데 필요한 파

라미터를 포함하는 JSON 객체를 생성합니다. 이 객체에는 액세스 키 ID와 사용자의 이름이 포함됩니다. deleteAccessKey 서비스 객체의 AWS.IAM 메서드를 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

var params = {
  AccessKeyId: "ACCESS_KEY_ID",
  UserName: "USER_NAME",
};

iam.deleteAccessKey(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node iam_deleteaccesskey.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

IAM 서버 인증서 작업

이 Node.js 코드 예제는 다음을 보여 줍니다.

- HTTPS 연결을 위해 서버 인증서를 관리하는 기본 작업을 수행하는 방법.

시나리오

AWS에서 웹 사이트나 애플리케이션에 대한 HTTPS 연결을 활성화하려면 SSL/TLS 서버 인증서가 필요합니다. 외부 공급자에게서 얻은 인증서를 AWS의 웹 사이트 또는 애플리케이션에서 사용하려면 해당 인증서를 IAM에 업로드하거나 AWS Certificate Manager로 가져와야 합니다.

이 예제에서는 일련의 Node.js 모듈을 사용하여 IAM에서 서버 인증서를 처리합니다. Node.js 모듈은 SDK for JavaScript로 `AWS.IAM` 클라이언트 클래스의 다음 메서드를 사용하여 서버 인증서를 관리합니다.

- listServerCertificates
- getServerCertificate
- updateServerCertificate
- deleteServerCertificate

IAM의 서버 인증서에 대한 자세한 내용은 IAM 사용 설명서의 서버 인증서 작업을 참조하세요.

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 이러한 작업들을 완료해야 합니다.

- Node.js를 설치합니다. Node.js 설치에 대한 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 공유 자격 증명 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.

서버 인증서 목록 표시

파일 이름이 `iam_listservercerts.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. IAM에 액세스하려면 `AWS.IAM` 서비스 객체를 생성합니다. `listServerCertificates` 서비스 객체의 `AWS.IAM` 메서드를 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });
```

```
iam.listServerCertificates({}, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node iam_listservercerts.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

서버 인증서 가져오기

파일 이름이 `iam_getservercert.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. IAM에 액세스하려면 `AWS.IAM` 서비스 객체를 생성합니다. 인증서를 가져오는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. 이 객체는 원하는 서버 인증서의 이름으로 구성됩니다. `getServerCertificates` 서비스 객체의 `AWS.IAM` 메서드를 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

iam.getServerCertificate(
  { ServerCertificateName: "CERTIFICATE_NAME" },
  function (err, data) {
    if (err) {
      console.log("Error", err);
    } else {
      console.log("Success", data);
    }
  }
);
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node iam_getservercert.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

서버 인증서 업데이트

파일 이름이 `iam_updateservercert.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. IAM에 액세스하려면 `AWS.IAM` 서비스 객체를 생성합니다. 인증서를 업데이트하는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. 이 객체는 기존 서버 인증서의 이름과 새 인증서의 이름으로 구성됩니다. `updateServerCertificate` 서비스 객체의 `AWS.IAM` 메서드를 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

var params = {
  ServerCertificateName: "CERTIFICATE_NAME",
  NewServerCertificateName: "NEW_CERTIFICATE_NAME",
};

iam.updateServerCertificate(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node iam_updateservercert.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

서버 인증서 삭제

파일 이름이 `iam_deleteservercert.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. IAM에 액세스하려면 `AWS.IAM` 서비스 객체를 생성합니다. 서버 인증서를 삭제하는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. 이 객체는 삭제하려는 인증서의 이름으로 구성됩니다. `deleteServerCertificates` 서비스 객체의 `AWS.IAM` 메서드를 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

iam.deleteServerCertificate(
  { ServerCertificateName: "CERTIFICATE_NAME" },
  function (err, data) {
    if (err) {
      console.log("Error", err);
    } else {
      console.log("Success", data);
    }
  }
);
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node iam_deleteservercert.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

IAM 계정 별칭 관리

이 Node.js 코드 예제는 다음을 보여 줍니다.

- AWS 계정 ID의 별칭을 관리하는 방법

시나리오

AWS 계정 ID 대신 회사 이름이나 기타 친숙한 식별자를 로그인 페이지의 URL에 포함하려는 경우 AWS 계정 ID의 별칭을 만들 수 있습니다. AWS 계정 별칭을 생성할 경우 명칭을 적용하기 위해 로그인 페이지 URL이 변경됩니다.

이 예제에서는 일련의 Node.js 모듈을 사용하여 IAM 계정 별칭을 생성하고 관리합니다. Node.js 모듈은 SDK for JavaScript로 `AWS.IAM` 클라이언트 클래스의 다음 메서드를 사용하여 별칭을 관리합니다.

- createAccountAlias
- listAccountAliases
- deleteAccountAlias

IAM 계정 별칭에 대한 자세한 내용은 AWS 계정 ID 및 별칭을 참조하세요.

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 이러한 작업들을 완료해야 합니다.

- Node.js를 설치합니다. Node.js 설치에 대한 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 공유 자격 증명 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.

계정 별칭 생성

파일 이름이 `iam_createaccountalias.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. IAM에 액세스하려면 `AWS.IAM` 서비스 객체를 생성합니다. 계정 별칭을 생성하는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. 이 객체에는 생성하려는 별칭이 포함됩니다. `createAccountAlias` 서비스 객체의 `AWS.IAM` 메서드를 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });
```

```
// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

iam.createAccountAlias({ AccountAlias: process.argv[2] }, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node iam_createaccountalias.js ALIAS
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

계정 별칭 나열

파일 이름이 `iam_listaccountaliases.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. IAM에 액세스하려면 `AWS.IAM` 서비스 객체를 생성합니다. 계정 별칭 목록을 표시하는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. 이 객체에는 반환할 항목의 최대 수가 포함되어 있습니다. `listAccountAliases` 서비스 객체의 `AWS.IAM` 메서드를 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

iam.listAccountAliases({ MaxItems: 10 }, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node iam_listaccountaliases.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

계정 별칭 삭제

파일 이름이 `iam_deleteaccountalias.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. IAM에 액세스하려면 `AWS.IAM` 서비스 객체를 생성합니다. 계정 별칭을 삭제하는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. 이 객체에는 삭제하려는 별칭이 포함됩니다. `deleteAccountAlias` 서비스 객체의 `AWS.IAM` 메서드를 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

iam.deleteAccountAlias({ AccountAlias: process.argv[2] }, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

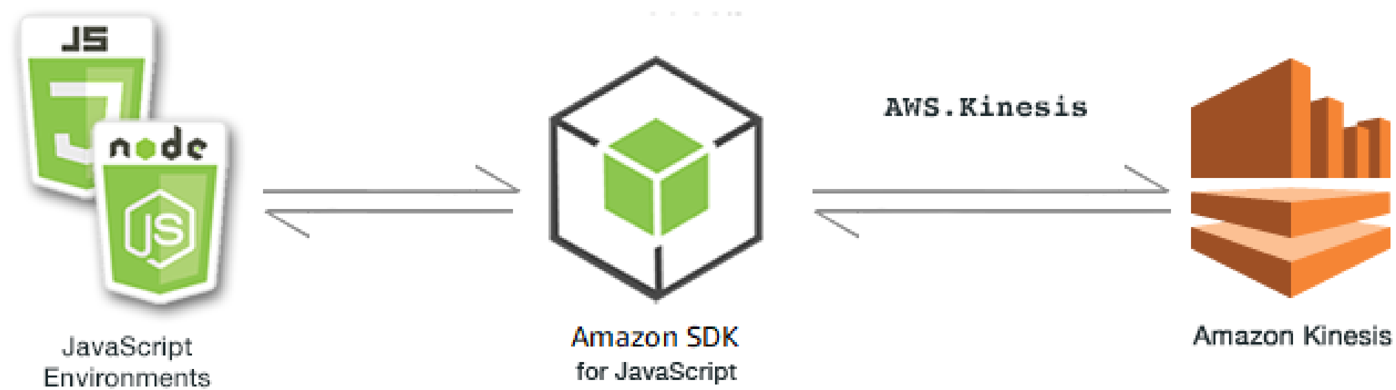
예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node iam_deleteaccountalias.js ALIAS
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

Amazon Kinesis 예제

Amazon Kinesis는 AWS의 스트리밍 데이터를 위한 플랫폼으로, 스트리밍 데이터를 로드하고 분석하기 위한 강력한 서비스를 제공하며 전문화된 요구에 맞게 사용자 지정 스트리밍 데이터 애플리케이션을 빌드할 수 있는 기능도 제공합니다.



Kinesis용 JavaScript API는 `AWS.Kinesis` 클라이언트 클래스를 통해 노출됩니다. Kinesis 클라이언트 클래스 사용에 대한 자세한 내용은 API 참조의 `Class: AWS.Kinesis` 섹션을 참조하세요.

주제

- Amazon Kinesis를 사용하여 웹 페이지 스크롤 진행 상황 캡처

Amazon Kinesis를 사용하여 웹 페이지 스크롤 진행 상황 캡처

이 브라우저 스크립트 예제는 다음을 보여 줍니다.

- 추후 분석을 위한 스트리밍 페이지 사용 지표의 예로 Amazon Kinesis를 사용하여 웹 페이지에서 스크롤 진행 상황을 캡처하는 방법

시나리오

이 예제에서는 단순 HTML 페이지에서 블로그 페이지의 콘텐츠를 시뮬레이션합니다. 독자가 시뮬레이션된 블로그 게시물을 스크롤하면 브라우저 스크립트는 SDK for JavaScript로 Kinesis 클라이언트 클래스의 `putRecords` 메서드를 사용하여 페이지 아래쪽 스크롤 거리를 기록하고 해당 데이터를 Kinesis에 전송합니다. 그러면 Amazon Kinesis Data Streams에서 캡처한 스트리밍 데이터를 Amazon EC2 인스턴스에서 처리하고 Amazon DynamoDB 및 Amazon Redshift를 포함한 여러 데이터 스토어에 저장할 수 있습니다.

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 이러한 작업들을 완료해야 합니다.

- Kinesis 스트림 생성 스트림의 리소스 ARN을 브라우저 스크립트에 포함시켜야 합니다. 자세한 내용은 Amazon Kinesis Data Streams 개발자 안내서의 Kinesis Streams란?을 참조하세요.
- 인증되지 않은 자격 증명에 대해 활성화된 액세스 권한이 있는 Amazon Cognito 자격 증명 풀 브라우저 스크립트에 대한 자격 증명을 획득하려면 자격 증명 풀 ID를 코드에 포함시켜야 합니다. Amazon Cognito 자격 증명 풀에 관한 자세한 내용은 Amazon Cognito 개발자 안내서의 자격 증명 풀을 참조하세요.
- 데이터를 Kinesis 스트림에 제출할 수 있는 권한을 부여하는 정책이 있는 Kinesis 역할을 생성합니다. IAM 역할 생성에 관한 자세한 내용은 IAM 사용 설명서의 AWS 서비스에 대한 권한을 위임할 역할 생성 섹션을 참조하세요.

IAM 역할을 생성할 때 다음 역할 정책을 사용합니다.

JSON

```
{
   "Version":"2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "mobileanalytics:PutEvents",
            "cognito-sync:*"
         ],
         "Resource": [
            "*"
         ]
      },
      {
         "Effect": "Allow",
         "Action": [
            "kinesis:Put*"
         ],
         "Resource": [
            "arn:aws:kinesis:us-east-1:111122223333:stream/stream-name"
         ]
      }
```

```
    ]
}
```

블로그 페이지

블로그 페이지용 HTML은 주로 `<div>` 요소 내에 포함된 일련의 단락으로 구성됩니다. 이 `<div>`의 스크롤 가능한 높이는 독자가 필요에 따라 콘텐츠를 스크롤한 거리를 계산하는 데 사용됩니다. HTML에는 `<script>` 요소 페어도 포함됩니다. 이러한 요소 중 하나는 SDK for JavaScript를 페이지에 추가하고 다른 하나는 페이지에서 스크롤 진행 상황을 캡처하여 Kinesis에 보고하는 브라우저 스크립트를 추가합니다.

```
<!DOCTYPE html>
<html>
    <head>
        <title>AWS SDK for JavaScript - Amazon Kinesis Application</title>
    </head>
    <body>
        <div id="BlogContent" style="width: 60%; height: 800px; overflow: auto;margin:
 auto; text-align: center;">
            <div>
                <p>
                Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vestibulum
 vitae nulla eget nisl bibendum feugiat. Fusce rhoncus felis at ultricies luctus.
 Vivamus fermentum cursus sem at interdum. Proin vel lobortis nulla. Aenean rutrum
 odio in tellus semper rhoncus. Nam eu felis ac augue dapibus laoreet vel in erat.
 Vivamus vitae mollis turpis. Integer sagittis dictum odio. Duis nec sapien diam.
 In imperdiet sem nec ante laoreet, vehicula facilisis sem placerat. Duis ut metus
 egestas, ullamcorper neque et, accumsan quam. Class aptent taciti sociosqu ad litora
 torquent per conubia nostra, per inceptos himenaeos.
                </p>
                <!-- Additional paragraphs in the blog page appear here -->
            </div>
        </div>
        <script src="https://sdk.amazonaws.com/js/aws-sdk-2.283.1.min.js"></script>
        <script src="kinesis-example.js"></script>
    </body>
</html>
```

SDK 구성

Amazon Cognito 자격 증명 풀 ID를 제공하는 `CognitoIdentityCredentials` 메서드를 직접 호출하여 SDK를 구성하는 데 필요한 자격 증명을 획득합니다. 성공하면 콜백 함수에서 Kinesis 서비스 객체를 생성합니다.

다음 코드 조각은 이 단계를 보여줍니다. (전체 예제는 웹 페이지 스크롤 진행 상황 코드 캡처 섹션을 참조하세요.)

```
// Configure Credentials to use Cognito
AWS.config.credentials = new AWS.CognitoIdentityCredentials({
  IdentityPoolId: "IDENTITY_POOL_ID",
});

AWS.config.region = "REGION";
// We're going to partition Amazon Kinesis records based on an identity.
// We need to get credentials first, then attach our event listeners.
AWS.config.credentials.get(function (err) {
  // attach event listener
  if (err) {
    alert("Error retrieving credentials.");
    console.error(err);
    return;
  }
  // create Amazon Kinesis service object
  var kinesis = new AWS.Kinesis({
    apiVersion: "2013-12-02",
  });
```

스크롤 레코드 생성

스크롤 진행 상황은 블로그 게시물의 콘텐츠를 포함하는 `scrollHeight`의 `scrollTop` 및 `<div>` 속성을 사용하여 계산됩니다. 각 스크롤 레코드는 `scroll` 이벤트에 대한 이벤트 리스너 함수에서 생성된 다음, Kinesis에 정기적으로 제출하기 위한 레코드 배열에 추가됩니다.

다음 코드 조각은 이 단계를 보여줍니다. (전체 예제는 웹 페이지 스크롤 진행 상황 코드 캡처 섹션을 참조하세요.)

```
  // Get the ID of the Web page element.
  var blogContent = document.getElementById("BlogContent");

  // Get Scrollable height
```

```
var scrollableHeight = blogContent.clientHeight;

var recordData = [];
var TID = null;
blogContent.addEventListener("scroll", function (event) {
  clearTimeout(TID);
  // Prevent creating a record while a user is actively scrolling
  TID = setTimeout(function () {
    // calculate percentage
    var scrollableElement = event.target;
    var scrollHeight = scrollableElement.scrollHeight;
    var scrollTop = scrollableElement.scrollTop;

    var scrollTopPercentage = Math.round((scrollTop / scrollHeight) * 100);
    var scrollBottomPercentage = Math.round(
      ((scrollTop + scrollableHeight) / scrollHeight) * 100
    );

    // Create the Amazon Kinesis record
    var record = {
      Data: JSON.stringify({
        blog: window.location.href,
        scrollTopPercentage: scrollTopPercentage,
        scrollBottomPercentage: scrollBottomPercentage,
        time: new Date(),
      }),
      PartitionKey: "partition-" + AWS.config.credentials.identityId,
    };
    recordData.push(record);
  }, 100);
});
```

Kinesis에 레코드 제출

배열에 레코드가 있는 경우 1초에 한 번씩 해당 보류 레코드가 Kinesis에 전송됩니다.

다음 코드 조각은 이 단계를 보여줍니다. (전체 예제는 웹 페이지 스크롤 진행 상황 코드 캡처 섹션을 참조하세요.)

```
// upload data to Amazon Kinesis every second if data exists
setInterval(function () {
  if (!recordData.length) {
    return;
```

```
    }
    // upload data to Amazon Kinesis
    kinesis.putRecords(
      {
        Records: recordData,
        StreamName: "NAME_OF_STREAM",
      },
      function (err, data) {
        if (err) {
          console.error(err);
        }
      }
    );
    // clear record data
    recordData = [];
  }, 1000);
});
```

웹 페이지 스크롤 진행 상황 코드 캡처

다음은 웹 페이지 스크롤 진행 상황 예제를 캡처하는 Kinesis용 브라우저 스크립트 코드입니다.

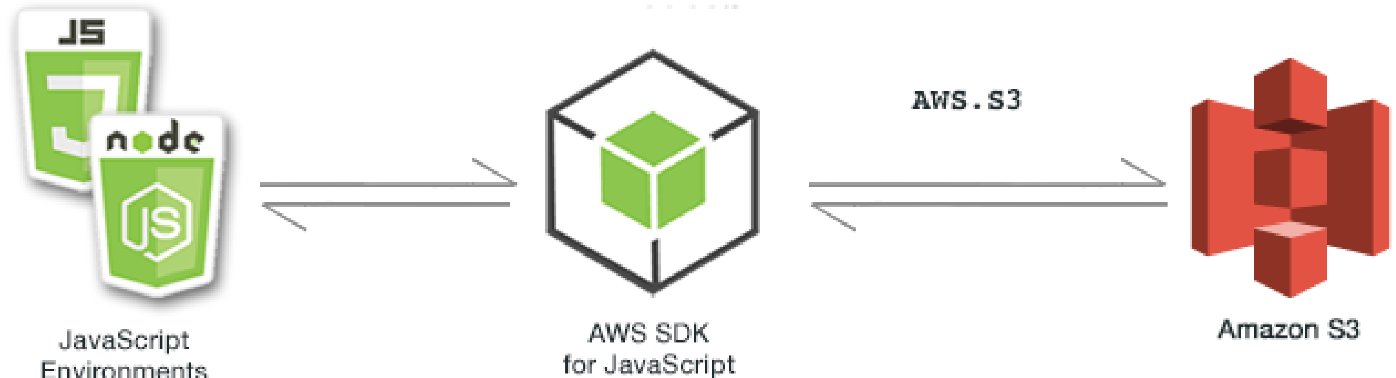
```
// Configure Credentials to use Cognito
AWS.config.credentials = new AWS.CognitoIdentityCredentials({
  IdentityPoolId: "IDENTITY_POOL_ID",
});

AWS.config.region = "REGION";
// We're going to partition Amazon Kinesis records based on an identity.
// We need to get credentials first, then attach our event listeners.
AWS.config.credentials.get(function (err) {
  // attach event listener
  if (err) {
    alert("Error retrieving credentials.");
    console.error(err);
    return;
  }
  // create Amazon Kinesis service object
  var kinesis = new AWS.Kinesis({
    apiVersion: "2013-12-02",
  });

  // Get the ID of the Web page element.
```

```
var blogContent = document.getElementById("BlogContent");

// Get Scrollable height
var scrollableHeight = blogContent.clientHeight;

var recordData = [];
var TID = null;
blogContent.addEventListener("scroll", function (event) {
  clearTimeout(TID);
  // Prevent creating a record while a user is actively scrolling
  TID = setTimeout(function () {
    // calculate percentage
    var scrollableElement = event.target;
    var scrollHeight = scrollableElement.scrollHeight;
    var scrollTop = scrollableElement.scrollTop;

    var scrollTopPercentage = Math.round((scrollTop / scrollHeight) * 100);
    var scrollBottomPercentage = Math.round(
      ((scrollTop + scrollableHeight) / scrollHeight) * 100
    );

    // Create the Amazon Kinesis record
    var record = {
      Data: JSON.stringify({
        blog: window.location.href,
        scrollTopPercentage: scrollTopPercentage,
        scrollBottomPercentage: scrollBottomPercentage,
        time: new Date(),
      }),
      PartitionKey: "partition-" + AWS.config.credentials.identityId,
    };
    recordData.push(record);
  }, 100);
});

// upload data to Amazon Kinesis every second if data exists
setInterval(function () {
  if (!recordData.length) {
    return;
  }
  // upload data to Amazon Kinesis
  kinesis.putRecords(
    {
      Records: recordData,
```

```
        StreamName: "NAME_OF_STREAM",
      },
      function (err, data) {
        if (err) {
          console.error(err);
        }
      }
    );
    // clear record data
    recordData = [];
  }, 1000);
});
```

Amazon S3 예제

Amazon Simple Storage Service(S3)는 확장성이 뛰어난 클라우드 스토리지를 제공하는 웹 서비스입니다. Amazon S3는 웹 어디에서나 원하는 양의 데이터를 저장하고 검색할 수 있는 간단한 웹 서비스 인터페이스를 갖춘 사용하기 쉬운 객체 스토리지입니다.



Amazon S3용 JavaScript API는 `AWS.S3` 클라이언트 클래스를 통해 노출됩니다. Amazon S3 클라이언트 클래스 사용에 대한 자세한 내용은 API 참조의 `Class: AWS.S3` 섹션을 참조하세요.

주제

- Amazon S3 브라우저 예제
- Amazon S3 Node.js 예제

Amazon S3 브라우저 예제

다음 주제는 브라우저에서 Amazon S3 버킷과의 상호 작용을 위해 AWS SDK for JavaScript가 사용되는 방식의 예제 두 가지를 설명합니다.

- 첫 번째 예제는 Amazon S3 버킷의 기존 사진은 모든 (미인증) 사용자가 볼 수 있는 단순한 시나리오를 보여줍니다.
- 두 번째 예제는 더욱 복잡한 시나리오로 사용자가 버킷의 사진에 대해 업로드, 삭제 등의 작업을 수행할 수 있습니다.

주제

- 브라우저에서 Amazon S3 버킷의 사진 보기
- 브라우저에서 Amazon S3에 사진 업로드

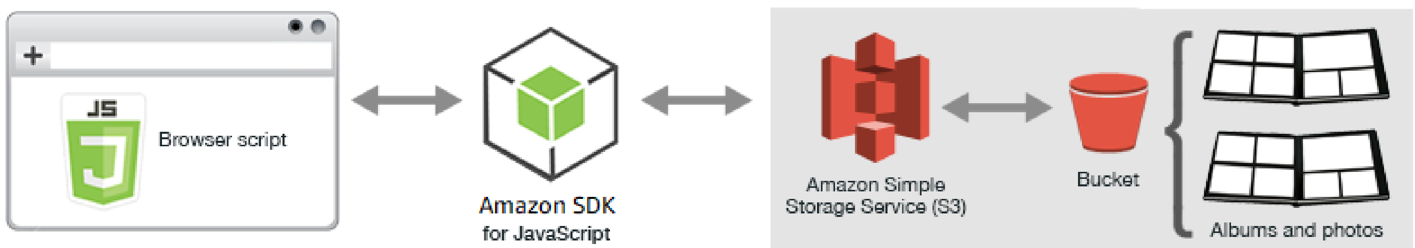
브라우저에서 Amazon S3 버킷의 사진 보기

이 브라우저 스크립트 코드 예제는 다음을 보여 줍니다.

- Amazon Simple Storage Service(S3) 버킷에서 사진 앨범을 생성하고 미인증 사용자가 사진을 볼 수 있도록 허용하는 방법

시나리오

이 예제에서는 단순한 HTML 페이지를 통해 사진 앨범의 사진을 볼 수 있는 브라우저 기반 애플리케이션이 제공됩니다. 사진 앨범은 사진이 업로드되는 Amazon S3 버킷에 있습니다.



브라우저 스크립트는 SDK for JavaScript를 사용하여 Amazon S3 버킷과 상호 작용합니다. 스크립트는 Amazon S3 클라이언트 클래스의 `listObjects` 메서드를 사용하여 사진 앨범을 볼 수 있게 합니다.

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 다음 작업을 완료합니다.

Note

이 예제에서는 Amazon S3 버킷과 Amazon Cognito 자격 증명 풀에 대해 동일한 AWS 리전을 사용해야 합니다.

버킷 생성

Amazon S3 콘솔에서 앨범과 사진을 저장할 수 있는 Amazon S3 버킷을 생성합니다. 콘솔을 사용한 S3 버킷 생성에 대한 자세한 내용을 알아보려면 Amazon Simple Storage Service 사용 설명서의 버킷 생성을 참조하세요.

S3 버킷 생성 시에는 다음을 수행합니다.

- 후속 사전 필수 작업인 역할 권한 구성 시 사용할 수 있도록 버킷 이름을 메모해 둡니다.
- 버킷을 생성할 AWS 지역을 선택합니다. 후속 사전 필수 작업인 자격 증명 풀 생성 시 Amazon Cognito 자격 증명 풀을 생성하는 데 사용할 리전과 동일한 리전이어야 합니다.
- Amazon Simple Storage Service 사용 설명서의 웹 사이트 액세스에 대한 권한 설정을 따라 버킷 권한을 구성합니다.

자격 증명 풀 생성

브라우저 스크립트에서 시작하기의 the section called “1단계: Amazon Cognito 자격 증명 풀 생성” 섹션을 참조하여 Amazon Cognito 콘솔에서 Amazon Cognito 자격 증명 풀을 생성합니다.

자격 증명 풀을 생성할 때는 인증되지 않은 자격 증명의 역할 이름과 자격 증명 풀의 이름을 메모해 둡니다.

역할 권한 구성

앨범 및 사진 보기를 허용하려면 방금 생성한 자격 증명 풀의 IAM 역할에 권한을 추가해야 합니다. 다음과 같이 정책을 생성하며 시작합니다.

1. IAM 콘솔을 엽니다.
2. 왼쪽 탐색 창에서 정책을 선택한 후 정책 생성 버튼을 선택합니다.
3. JSON 탭에 다음의 JSON 정의를 입력하되, BUCKET_NAME은 버킷 이름으로 대체합니다.

   JSON

```
{
   "Version":"2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "s3:ListBucket"
         ],
         "Resource": [
            "arn:aws:s3:::BUCKET_NAME"
         ]
      }
   ]
}
```

4. 정책 검토 버튼을 선택하고 정책 이름을 지정한 다음 설명을 입력하고(원할 경우), 정책 생성 버튼을 선택합니다.

   나중에 찾아서 IAM 역할에 연결할 수 있도록 이름을 메모해 둡니다.

정책이 생성되면 IAM 콘솔로 돌아갑니다. 이전 사전 필수 작업인 자격 증명 풀 생성에서 Amazon Cognito가 생성한 미인증 자격 증명의 IAM 역할을 찾습니다. 방금 생성한 정책을 사용하여 이 자격 증명에 권한을 추가합니다.

이 작업의 워크플로는 일반적으로 브라우저 스크립트에서 시작하기의 the section called “2단계: 생성된 IAM 역할에 정책 추가” 섹션과 동일하지만 몇 가지 차이점이 있습니다.

- Amazon Polly에 대한 정책이 아닌 방금 생성한 새로운 정책을 사용합니다.
- 권한 연결 페이지에서 새로운 정책을 빠르게 찾아 보려면 필터 정책 목록을 열고 고객 관리형을 선택합니다.

IAM 역할 생성에 관한 추가 정보는 IAM 사용 설명서의 AWS 서비스에 대한 권한을 위임할 역할 생성 섹션을 참조하세요.

CORS 구성

브라우저 스크립트가 Amazon 버킷에 액세스하려면 다음과 같이 CORS 구성을 설정해야 합니다.

Important

새 S3 콘솔에서 CORS 구성은 JSON이어야 합니다.

JSON

```
[
    {
        "AllowedHeaders": [
            "*"
        ],
        "AllowedMethods": [
            "HEAD",
            "GET"
        ],
        "AllowedOrigins": [
            "*"
        ]
    }
]
```

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<CORSConfiguration xmlns="http://s3.amazonaws.com/doc/2006-03-01/">
    <CORSRule>
        <AllowedOrigin>*</AllowedOrigin>
        <AllowedMethod>GET</AllowedMethod>
        <AllowedMethod>HEAD</AllowedMethod>
        <AllowedHeader>*</AllowedHeader>
    </CORSRule>
</CORSConfiguration>
```

앨범 생성 및 사진 업로드

이 예제에서는 사용자가 이미 버킷에 있는 사진만 볼 수 있기 때문에 버킷에 앨범을 생성하고 그 앨범으로 사진을 업로드해야 합니다.

Note

이 예제의 경우, 사진 파일의 파일명은 하나의 밑줄("_")로 시작해야 합니다. 이 문자는 나중에 필터링하는 데 중요합니다. 또한, 사진 소유자의 저작권을 존중해 주십시오.

1. Amazon S3 콘솔에서 앞서 생성한 버킷을 엽니다.
2. 개요 탭에서 폴더 만들기 버튼을 선택하여 폴더를 생성합니다. 이 예제에서는 폴더명을 "album1", "album2" 및 "album3"으로 지정합니다.
3. album1과 album2에 대해 폴더를 선택한 다음 다음과 같이 사진을 업로드합니다.
   a. 업로드 버튼을 선택합니다.
   b. 사용할 사진 파일을 끌거나 선택하고 다음을 선택합니다.
   c. 퍼블릭 권한 관리에서 이 객체에 퍼블릭 읽기 액세스 권한을 부여함을 선택합니다.
   d. 업로드 버튼(왼쪽 아래 모서리)을 선택합니다.
4. album3은 비워둡니다.

웹페이지 정의

사진 보기 애플리케이션을 위한 HTML은 브라우저 스크립트가 보기 인터페이스를 생성하는 `<div>` 요소로 구성됩니다. 첫 번째 `<script>` 요소는 SDK를 브라우저 스크립트에 추가합니다. 두 번째 `<script>` 요소는 브라우저 스크립트 코드를 담는 외부 JavaScript 파일을 추가합니다.

이 예제에서는 파일명이 `PhotoViewer.js`로 지정되었으며 HTML 파일과 동일한 폴더에 위치합니다. AWS SDK for JavaScript API 참조 가이드의 SDK for JavaScript에 대한 API 참조 섹션에서 현재 SDK_VERSION_NUMBER를 찾을 수 있습니다.

```
<!DOCTYPE html>
<html>
  <head>
    <!-- **DO THIS**: -->
    <!--   Replace SDK_VERSION_NUMBER with the current SDK version number -->
    <script src="https://sdk.amazonaws.com/js/aws-sdk-SDK_VERSION_NUMBER.js"></script>
```

```
    <script src="./PhotoViewer.js"></script>
    <script>listAlbums();</script>
  </head>
  <body>
    <h1>Photo Album Viewer</h1>
    <div id="viewer" />
  </body>
</html>
```

SDK 구성

`CognitoIdentityCredentials` 메서드를 호출하여 SDK를 구성하는 데 필요한 자격 증명을 획득합니다. Amazon Cognito 자격 증명 풀 ID를 제공해야 합니다. 그런 다음, `AWS.S3` 서비스 객체를 생성합니다.

```
// **DO THIS**:
//   Replace BUCKET_NAME with the bucket name.
//
var albumBucketName = "BUCKET_NAME";

// **DO THIS**:
//   Replace this block of code with the sample code located at:
//   Cognito -- Manage Identity Pools -- [identity_pool_name] -- Sample Code --
 JavaScript
//
// Initialize the Amazon Cognito credentials provider
AWS.config.region = "REGION"; // Region
AWS.config.credentials = new AWS.CognitoIdentityCredentials({
  IdentityPoolId: "IDENTITY_POOL_ID",
});

// Create a new service object
var s3 = new AWS.S3({
  apiVersion: "2006-03-01",
  params: { Bucket: albumBucketName },
});

// A utility function to create HTML.
function getHtml(template) {
  return template.join("\n");
}
```

이 예제에 있는 코드의 나머지 부분은 버킷의 앨범과 사진에 관한 정보를 수집 및 제공하기 위해 다음의 함수를 정의합니다.

- `listAlbums`
- `viewAlbum`

버킷에 있는 앨범 목록 표시

버킷에 있는 모든 앨범의 목록을 표시하기 위해 애플리케이션의 `listAlbums` 함수는 `listObjects` 서비스 객체의 `AWS.S3` 메서드를 호출합니다. 이 함수는 `CommonPrefixes` 속성을 사용하므로 이 호출은 앨범으로 사용된 객체(폴더)만 반환합니다.

이 함수의 나머지 부분은 Amazon S3 버킷에서 앨범 목록을 가져오고 웹 페이지에 앨범 목록을 표시하는 데 필요한 HTML을 생성합니다.

```
// List the photo albums that exist in the bucket.
function listAlbums() {
  s3.listObjects({ Delimiter: "/" }, function (err, data) {
    if (err) {
      return alert("There was an error listing your albums: " + err.message);
    } else {
      var albums = data.CommonPrefixes.map(function (commonPrefix) {
        var prefix = commonPrefix.Prefix;
        var albumName = decodeURIComponent(prefix.replace("/", ""));
        return getHtml([
          "<li>",
          '<button style="margin:5px;" onclick="viewAlbum(\'' +
            albumName +
            "')\">",
          albumName,
          "</button>",
          "</li>",
        ]);
      });
      var message = albums.length
        ? getHtml(["<p>Click on an album name to view it.</p>"])
        : "<p>You do not have any albums. Please Create album.";
      var htmlTemplate = [
        "<h2>Albums</h2>",
        message,
        "<ul>",
        getHtml(albums),
```

```
        "</ul>",
      ];
      document.getElementById("viewer").innerHTML = getHtml(htmlTemplate);
    }
  });
}
```

앨범 보기

Amazon S3 버킷에 앨범의 콘텐츠를 표시하기 위해 애플리케이션의 `viewAlbum` 함수는 앨범 이름을 가져오고 해당 앨범에 대한 Amazon S3 키를 생성합니다. 그런 다음 이 함수는 `listObjects` 서비스 객체의 `AWS.S3` 메서드를 호출하여 앨범에 있는 모든 객체(사진)의 목록을 획득합니다.

이 함수의 나머지 부분은 앨범에 있는 객체 목록을 가져오고 웹 페이지에 사진을 표시하는 데 필요한 HTML을 생성합니다.

```
// Show the photos that exist in an album.
function viewAlbum(albumName) {
  var albumPhotosKey = encodeURIComponent(albumName) + "/";
  s3.listObjects({ Prefix: albumPhotosKey }, function (err, data) {
    if (err) {
      return alert("There was an error viewing your album: " + err.message);
    }
    // 'this' references the AWS.Request instance that represents the response
    var href = this.request.httpRequest.endpoint.href;
    var bucketUrl = href + albumBucketName + "/";

    var photos = data.Contents.map(function (photo) {
      var photoKey = photo.Key;
      var photoUrl = bucketUrl + encodeURIComponent(photoKey);
      return getHtml([
        "<span>",
        "<div>",
        "<br/>",
        '<img style="width:128px;height:128px;" src="' + photoUrl + '"/>',
        "</div>",
        "<div>",
        "<span>",
        photoKey.replace(albumPhotosKey, ""),
        "</span>",
        "</div>",
        "</span>",
      ]);
```

```
    });
    var message = photos.length
      ? "<p>The following photos are present.</p>"
      : "<p>There are no photos in this album.</p>";
    var htmlTemplate = [
      "<div>",
      '<button onclick="listAlbums()">',
      "Back To Albums",
      "</button>",
      "</div>",
      "<h2>",
      "Album: " + albumName,
      "</h2>",
      message,
      "<div>",
      getHtml(photos),
      "</div>",
      "<h2>",
      "End of Album: " + albumName,
      "</h2>",
      "<div>",
      '<button onclick="listAlbums()">',
      "Back To Albums",
      "</button>",
      "</div>",
    ];
    document.getElementById("viewer").innerHTML = getHtml(htmlTemplate);
    document
      .getElementsByTagName("img")[0]
      .setAttribute("style", "display:none;");
  });
}
```

Amazon S3 버킷의 사진 보기: 전체 코드

이 섹션에는 Amazon S3 버킷의 사진을 볼 수 있는 예제를 위한 HTML 및 JavaScript 전체 코드가 포함되어 있습니다. 자세한 내용과 사전 조건은 상위 섹션을 참조하세요.

HTML 예제:

```
<!DOCTYPE html>
<html>
  <head>
    <!-- **DO THIS**: -->
```

```
    <!--   Replace SDK_VERSION_NUMBER with the current SDK version number -->
    <script src="https://sdk.amazonaws.com/js/aws-sdk-SDK_VERSION_NUMBER.js"></script>
    <script src="./PhotoViewer.js"></script>
    <script>listAlbums();</script>
  </head>
  <body>
    <h1>Photo Album Viewer</h1>
    <div id="viewer" />
  </body>
</html>
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

브라우저 스크립트 코드 예제:

```
//
// Data constructs and initialization.
//

// **DO THIS**:
//   Replace BUCKET_NAME with the bucket name.
//
var albumBucketName = "BUCKET_NAME";

// **DO THIS**:
//   Replace this block of code with the sample code located at:
//   Cognito -- Manage Identity Pools -- [identity_pool_name] -- Sample Code --
 JavaScript
//
// Initialize the Amazon Cognito credentials provider
AWS.config.region = "REGION"; // Region
AWS.config.credentials = new AWS.CognitoIdentityCredentials({
  IdentityPoolId: "IDENTITY_POOL_ID",
});

// Create a new service object
var s3 = new AWS.S3({
  apiVersion: "2006-03-01",
  params: { Bucket: albumBucketName },
});

// A utility function to create HTML.
function getHtml(template) {
  return template.join("\n");
```

```
}

//
// Functions
//

// List the photo albums that exist in the bucket.
function listAlbums() {
  s3.listObjects({ Delimiter: "/" }, function (err, data) {
    if (err) {
      return alert("There was an error listing your albums: " + err.message);
    } else {
      var albums = data.CommonPrefixes.map(function (commonPrefix) {
        var prefix = commonPrefix.Prefix;
        var albumName = decodeURIComponent(prefix.replace("/", ""));
        return getHtml([
          "<li>",
          '<button style="margin:5px;" onclick="viewAlbum(\'' +
            albumName +
            "')\">",
          albumName,
          "</button>",
          "</li>",
        ]);
      });
      var message = albums.length
        ? getHtml(["<p>Click on an album name to view it.</p>"])
        : "<p>You do not have any albums. Please Create album.";
      var htmlTemplate = [
        "<h2>Albums</h2>",
        message,
        "<ul>",
        getHtml(albums),
        "</ul>",
      ];
      document.getElementById("viewer").innerHTML = getHtml(htmlTemplate);
    }
  });
}

// Show the photos that exist in an album.
function viewAlbum(albumName) {
  var albumPhotosKey = encodeURIComponent(albumName) + "/";
  s3.listObjects({ Prefix: albumPhotosKey }, function (err, data) {
```

```
    if (err) {
      return alert("There was an error viewing your album: " + err.message);
    }
    // 'this' references the AWS.Request instance that represents the response
    var href = this.request.httpRequest.endpoint.href;
    var bucketUrl = href + albumBucketName + "/";

    var photos = data.Contents.map(function (photo) {
      var photoKey = photo.Key;
      var photoUrl = bucketUrl + encodeURIComponent(photoKey);
      return getHtml([
        "<span>",
        "<div>",
        "<br/>",
        '<img style="width:128px;height:128px;" src="' + photoUrl + '"/>',
        "</div>",
        "<div>",
        "<span>",
        photoKey.replace(albumPhotosKey, ""),
        "</span>",
        "</div>",
        "</span>",
      ]);
    });
    var message = photos.length
      ? "<p>The following photos are present.</p>"
      : "<p>There are no photos in this album.</p>";
    var htmlTemplate = [
      "<div>",
      '<button onclick="listAlbums()">',
      "Back To Albums",
      "</button>",
      "</div>",
      "<h2>",
      "Album: " + albumName,
      "</h2>",
      message,
      "<div>",
      getHtml(photos),
      "</div>",
      "<h2>",
      "End of Album: " + albumName,
      "</h2>",
      "<div>",
```

```
        '<button onclick="listAlbums()">',
        "Back To Albums",
        "</button>",
        "</div>",
      ];
      document.getElementById("viewer").innerHTML = getHtml(htmlTemplate);
      document
        .getElementsByTagName("img")[0]
        .setAttribute("style", "display:none;");
    });
  }
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

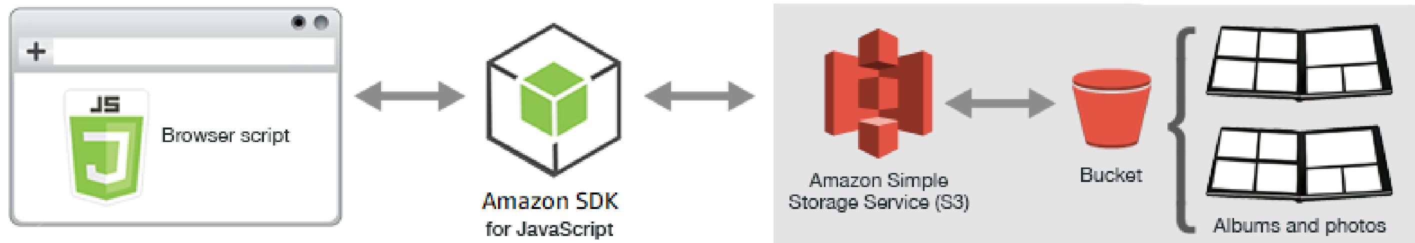
브라우저에서 Amazon S3에 사진 업로드

이 브라우저 스크립트 코드 예제는 다음을 보여 줍니다.

- 사용자가 Amazon S3 버킷에서 사진 앨범을 생성하고 해당 앨범으로 사진을 업로드할 수 있도록 브라우저 애플리케이션을 생성하는 방법

시나리오

이 예제에서는 간단한 HTML 페이지에서 사진을 업로드할 수 있는 Amazon S3 버킷의 사진 앨범을 생성하기 위한 브라우저 기반 애플리케이션을 제공합니다. 이 애플리케이션을 사용하여 추가한 사진과 앨범을 삭제할 수 있습니다.



브라우저 스크립트는 SDK for JavaScript를 사용하여 Amazon S3 버킷과 상호 작용합니다. Amazon S3 클라이언트 클래스의 다음 메서드를 사용하여 사진 앨범 애플리케이션을 활성화합니다.

- `listObjects`
- `headObject`
- `putObject`
- `upload`
- `deleteObject`
- `deleteObjects`

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 이러한 작업들을 완료해야 합니다.

- Amazon S3 콘솔에서 앨범에 사진을 저장하는 데 사용할 Amazon S3 버킷을 생성합니다. 콘솔 내 버킷 생성에 대한 자세한 내용을 알아보려면 Amazon Simple Storage Service 사용 설명서의 버킷 생성을 참조하세요. 객체에 대한 읽기 및 쓰기 권한이 모두 있는지 확인합니다. 버킷 설정 권한에 대한 자세한 내용은 웹 사이트 액세스 권한 설정을 참조하세요.
- Amazon Cognito 콘솔에서 Amazon S3 버킷과 동일한 리전에 있는 미인증 사용자에 대해 활성화된 액세스 권한이 있는 연동 자격 증명을 사용하여 Amazon Cognito 자격 증명 풀을 생성합니다. 브라우저 스크립트에 대한 자격 증명을 획득하려면 자격 증명 풀 ID를 코드에 포함시켜야 합니다. 자세한 내용은 Amazon Cognito 개발자 안내서의 Amazon Cognito 자격 증명 풀(페더레이션 자격 증명)을 참조하세요.
- IAM 콘솔에서 인증되지 않은 사용자에 대해 Amazon Cognito가 생성한 IAM 역할을 찾습니다. 다음 정책을 추가하여 읽기 및 쓰기 권한을 Amazon S3 버킷에 부여합니다. IAM 역할 생성에 관한 자세한 내용은 IAM 사용 설명서의 AWS 서비스에 대한 권한을 위임할 역할 생성 섹션을 참조하세요.

  인증되지 않은 사용자에 대해 Amazon Cognito가 생성한 IAM 역할에 이 역할 정책을 사용합니다.

  Warning

  인증되지 않은 사용자의 액세스를 활성화하면 버킷과 해당 버킷의 모든 객체, 전 세계 모든 사람에게 쓰기 권한을 부여하게 됩니다. 이러한 보안 태세는 이 사례의 기본 목표에 초점을 맞추는 데 유용합니다. 그러나 실제 사례의 경우, 인증된 사용자 및 객체 소유권의 사용 같이 더욱 강화된 보안을 사용하는 것이 좋습니다.

JSON

```
{
   "Version":"2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "s3:DeleteObject",
            "s3:GetObject",
            "s3:ListBucket",
            "s3:PutObject",
            "s3:PutObjectAcl"
         ],
         "Resource": [
            "arn:aws:s3:::BUCKET_NAME",
            "arn:aws:s3:::BUCKET_NAME/*"
         ]
      }
   ]
}
```

CORS 구성

브라우저 스크립트가 Amazon S3 버킷에 액세스하려면 먼저 다음과 같이 CORS 구성을 설정해야 합니다.

> **Important**
> 새 S3 콘솔에서 CORS 구성은 JSON이어야 합니다.

JSON

```
[
    {
        "AllowedHeaders": [
            "*"
        ],
```

```
        "AllowedMethods": [
            "HEAD",
            "GET",
            "PUT",
            "POST",
            "DELETE"
        ],
        "AllowedOrigins": [
            "*"
        ],
        "ExposeHeaders": [
            "ETag"
        ]
    }
]
```

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<CORSConfiguration xmlns="http://s3.amazonaws.com/doc/2006-03-01/">
    <CORSRule>
        <AllowedOrigin>*</AllowedOrigin>
        <AllowedMethod>POST</AllowedMethod>
        <AllowedMethod>GET</AllowedMethod>
        <AllowedMethod>PUT</AllowedMethod>
        <AllowedMethod>DELETE</AllowedMethod>
        <AllowedMethod>HEAD</AllowedMethod>
        <AllowedHeader>*</AllowedHeader>
        <ExposeHeader>ETag</ExposeHeader>
    </CORSRule>
</CORSConfiguration>
```

웹 페이지

사진 업로드 애플리케이션을 위한 HTML은 브라우저 스크립트가 업로드 사용자 인터페이스를 생성하는 <div> 요소로 구성됩니다. 첫 번째 <script> 요소는 SDK를 브라우저 스크립트에 추가합니다. 두 번째 <script> 요소는 브라우저 스크립트 코드를 담는 외부 JavaScript 파일을 추가합니다.

```
<!DOCTYPE html>
<html>
  <head>
```

```
      <!-- **DO THIS**: -->
    <!--   Replace SDK_VERSION_NUMBER with the current SDK version number -->
    <script src="https://sdk.amazonaws.com/js/aws-sdk-SDK_VERSION_NUMBER.js"></script>
    <script src="./s3_photoExample.js"></script>
    <script>
       function getHtml(template) {
          return template.join('\n');
       }
       listAlbums();
    </script>
  </head>
  <body>
    <h1>My Photo Albums App</h1>
    <div id="app"></div>
  </body>
</html>
```

SDK 구성

Amazon Cognito 자격 증명 풀 ID를 제공하는 `CognitoIdentityCredentials` 메서드를 직접 호출하여 SDK를 구성하는 데 필요한 자격 증명을 획득합니다. 다음에는, `AWS.S3` 서비스 객체를 생성합니다.

```
var albumBucketName = "BUCKET_NAME";
var bucketRegion = "REGION";
var IdentityPoolId = "IDENTITY_POOL_ID";

AWS.config.update({
  region: bucketRegion,
  credentials: new AWS.CognitoIdentityCredentials({
    IdentityPoolId: IdentityPoolId,
  }),
});

var s3 = new AWS.S3({
  apiVersion: "2006-03-01",
  params: { Bucket: albumBucketName },
});
```

이 예제의 나머지 거의 모든 코드는 버킷의 앨범에 대한 정보를 수집 및 제공하고, 앨범에 사진을 업로드하고 업로드된 사진을 표시하며, 사진과 앨범을 삭제하는 일련의 함수로 구성됩니다. 이러한 함수는 다음과 같습니다.

- `listAlbums`
- `createAlbum`
- `viewAlbum`
- `addPhoto`
- `deleteAlbum`
- `deletePhoto`

버킷에 있는 앨범 목록 표시

이 애플리케이션은 Amazon S3 버킷의 앨범을 객체로 생성합니다. 이 객체의 키는 객체 함수가 폴더임을 표시하는 슬래시 문자로 시작합니다. 버킷에 있는 모든 앨범의 목록을 표시하기 위해 애플리케이션의 `listAlbums` 함수는 `listObjects`를 사용하면서 `AWS.S3` 서비스 객체의 `commonPrefix` 메서드를 호출합니다. 따라서 이 호출은 앨범으로 사용되는 객체만 반환합니다.

이 함수의 나머지 부분은 Amazon S3 버킷에서 앨범 목록을 가져오고 웹 페이지에 앨범 목록을 표시하는 데 필요한 HTML을 생성합니다. 또한 개별 앨범 삭제 및 열기도 활성화합니다.

```
function listAlbums() {
  s3.listObjects({ Delimiter: "/" }, function (err, data) {
    if (err) {
      return alert("There was an error listing your albums: " + err.message);
    } else {
      var albums = data.CommonPrefixes.map(function (commonPrefix) {
        var prefix = commonPrefix.Prefix;
        var albumName = decodeURIComponent(prefix.replace("/", ""));
        return getHtml([
          "<li>",
          "<span onclick=\"deleteAlbum('" + albumName + "')\">X</span>",
          "<span onclick=\"viewAlbum('" + albumName + "')\">",
          albumName,
          "</span>",
          "</li>",
        ]);
      });
      var message = albums.length
        ? getHtml([
            "<p>Click on an album name to view it.</p>",
            "<p>Click on the X to delete the album.</p>",
          ])
        : "<p>You do not have any albums. Please Create album.";
```

```
      var htmlTemplate = [
        "<h2>Albums</h2>",
        message,
        "<ul>",
        getHtml(albums),
        "</ul>",
        "<button onclick=\"createAlbum(prompt('Enter Album Name:'))\">",
        "Create New Album",
        "</button>",
      ];
      document.getElementById("app").innerHTML = getHtml(htmlTemplate);
    }
  });
}
```

버킷에서 앨범 생성

Amazon S3 버킷에서 앨범을 생성하기 위해 애플리케이션의 `createAlbum` 함수는 새 앨범에 지정된 이름을 확인하여 적합한 문자가 이름에 포함되는지 확인합니다. 그런 다음 함수는 Amazon S3 객체 키를 구성하여 Amazon S3 서비스 객체의 `headObject` 메서드에 전달합니다. 이 메서드는 지정된 키에 대한 메타데이터를 반환하므로, 데이터를 반환하는 경우 해당 키가 있는 객체가 이미 있는 것입니다.

앨범이 아직 없는 경우 이 함수는 `putObject` 서비스 객체의 `AWS.S3`메서드를 호출하여 앨범을 생성합니다. 그런 다음 `viewAlbum` 함수를 호출하여 비어 있는 새 앨범을 표시합니다.

```
function createAlbum(albumName) {
  albumName = albumName.trim();
  if (!albumName) {
    return alert("Album names must contain at least one non-space character.");
  }
  if (albumName.indexOf("/") !== -1) {
    return alert("Album names cannot contain slashes.");
  }
  var albumKey = encodeURIComponent(albumName);
  s3.headObject({ Key: albumKey }, function (err, data) {
    if (!err) {
      return alert("Album already exists.");
    }
    if (err.code !== "NotFound") {
      return alert("There was an error creating your album: " + err.message);
    }
    s3.putObject({ Key: albumKey }, function (err, data) {
      if (err) {
```

```
        return alert("There was an error creating your album: " + err.message);
      }
      alert("Successfully created album.");
      viewAlbum(albumName);
    });
  });
}
```

앨범 보기

Amazon S3 버킷에 앨범의 콘텐츠를 표시하기 위해 애플리케이션의 `viewAlbum` 함수는 앨범 이름을 가져오고 해당 앨범에 대한 Amazon S3 키를 생성합니다. 그런 다음 이 함수는 `listObjects` 서비스 객체의 `AWS.S3` 메서드를 호출하여 앨범에 있는 모든 객체(사진)의 목록을 획득합니다.

이 함수의 나머지 부분은 앨범에서 객체(사진) 목록을 가져오고 웹 페이지에 사진을 표시하는 데 필요한 HTML을 생성합니다. 또한 개별 사진을 삭제하고 앨범 목록으로 다시 이동하는 작업도 활성화합니다.

```
function viewAlbum(albumName) {
  var albumPhotosKey = encodeURIComponent(albumName) + "/";
  s3.listObjects({ Prefix: albumPhotosKey }, function (err, data) {
    if (err) {
      return alert("There was an error viewing your album: " + err.message);
    }
    // 'this' references the AWS.Response instance that represents the response
    var href = this.request.httpRequest.endpoint.href;
    var bucketUrl = href + albumBucketName + "/";

    var photos = data.Contents.map(function (photo) {
      var photoKey = photo.Key;
      var photoUrl = bucketUrl + encodeURIComponent(photoKey);
      return getHtml([
        "<span>",
        "<div>",
        '<img style="width:128px;height:128px;" src="' + photoUrl + '"/>',
        "</div>",
        "<div>",
        "<span onclick=\"deletePhoto('" +
          albumName +
          "','" +
          photoKey +
          "')\">",
        "X",
```

```
        "</span>",
        "<span>",
        photoKey.replace(albumPhotosKey, ""),
        "</span>",
        "</div>",
        "</span>",
      ]);
    });
    var message = photos.length
      ? "<p>Click on the X to delete the photo</p>"
      : "<p>You do not have any photos in this album. Please add photos.</p>";
    var htmlTemplate = [
      "<h2>",
      "Album: " + albumName,
      "</h2>",
      message,
      "<div>",
      getHtml(photos),
      "</div>",
      '<input id="photoupload" type="file" accept="image/*">',
      '<button id="addphoto" onclick="addPhoto(\'' + albumName + "')\">",
      "Add Photo",
      "</button>",
      '<button onclick="listAlbums()">',
      "Back To Albums",
      "</button>",
    ];
    document.getElementById("app").innerHTML = getHtml(htmlTemplate);
  });
}
```

앨범에 사진 추가

애플리케이션의 addPhoto 함수가 Amazon S3 버킷의 앨범에 사진을 업로드하기 위해 웹 페이지에서 파일 선택기 요소를 사용하여 업로드할 파일을 식별합니다. 그런 현재 앨범 이름과 파일 이름에서 업로드할 사진에 대한 키를 구성합니다.

이 함수는 Amazon S3 서비스 객체의 `upload` 메서드를 호출하여 사진을 업로드합니다. 사진을 업로드한 후 이 함수는 업로드된 사진이 나타나도록 앨범을 다시 표시합니다.

```
function addPhoto(albumName) {
  var files = document.getElementById("photoupload").files;
  if (!files.length) {
```

```
    return alert("Please choose a file to upload first.");
  }
  var file = files[0];
  var fileName = file.name;
  var albumPhotosKey = encodeURIComponent(albumName) + "/";

  var photoKey = albumPhotosKey + fileName;

  // Use S3 ManagedUpload class as it supports multipart uploads
  var upload = new AWS.S3.ManagedUpload({
    params: {
      Bucket: albumBucketName,
      Key: photoKey,
      Body: file,
    },
  });

  var promise = upload.promise();

  promise.then(
    function (data) {
      alert("Successfully uploaded photo.");
      viewAlbum(albumName);
    },
    function (err) {
      return alert("There was an error uploading your photo: ", err.message);
    }
  );
}
```

사진 삭제

애플리케이션의 `deletePhoto` 함수가 Amazon S3 버킷의 앨범에서 사진을 삭제하기 위해 Amazon S3 서비스 객체의 `deleteObject` 메서드를 호출합니다. 이렇게 하면 함수에 전달된 `photoKey` 값으로 지정되는 사진이 삭제됩니다.

```
function deletePhoto(albumName, photoKey) {
  s3.deleteObject({ Key: photoKey }, function (err, data) {
    if (err) {
      return alert("There was an error deleting your photo: ", err.message);
    }
    alert("Successfully deleted photo.");
    viewAlbum(albumName);
```

```
  });
}
```

앨범 삭제

애플리케이션의 `deleteAlbum` 함수가 Amazon S3 버킷의 앨범을 삭제하기 위해 Amazon S3 서비스 객체의 `deleteObjects` 메서드를 호출합니다.

```
function deleteAlbum(albumName) {
  var albumKey = encodeURIComponent(albumName) + "/";
  s3.listObjects({ Prefix: albumKey }, function (err, data) {
    if (err) {
      return alert("There was an error deleting your album: ", err.message);
    }
    var objects = data.Contents.map(function (object) {
      return { Key: object.Key };
    });
    s3.deleteObjects(
      {
        Delete: { Objects: objects, Quiet: true },
      },
      function (err, data) {
        if (err) {
          return alert("There was an error deleting your album: ", err.message);
        }
        alert("Successfully deleted album.");
        listAlbums();
      }
    );
  });
}
```

Amazon S3에 사진 업로드: 전체 코드

이 섹션에는 사진이 Amazon S3 사진 앨범에 업로드되는 예제에 관한 전체 HTML 및 JavaScript 코드가 포함되어 있습니다. 자세한 내용과 사전 조건은 상위 섹션을 참조하세요.

HTML 예제:

```
<!DOCTYPE html>
<html>
  <head>
     <!-- **DO THIS**: -->
```

```
    <!--   Replace SDK_VERSION_NUMBER with the current SDK version number -->
    <script src="https://sdk.amazonaws.com/js/aws-sdk-SDK_VERSION_NUMBER.js"></script>
    <script src="./s3_photoExample.js"></script>
    <script>
       function getHtml(template) {
          return template.join('\n');
       }
       listAlbums();
    </script>
  </head>
  <body>
    <h1>My Photo Albums App</h1>
    <div id="app"></div>
  </body>
</html>
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

브라우저 스크립트 코드 예제:

```
var albumBucketName = "BUCKET_NAME";
var bucketRegion = "REGION";
var IdentityPoolId = "IDENTITY_POOL_ID";

AWS.config.update({
  region: bucketRegion,
  credentials: new AWS.CognitoIdentityCredentials({
    IdentityPoolId: IdentityPoolId,
  }),
});

var s3 = new AWS.S3({
  apiVersion: "2006-03-01",
  params: { Bucket: albumBucketName },
});

function listAlbums() {
  s3.listObjects({ Delimiter: "/" }, function (err, data) {
    if (err) {
      return alert("There was an error listing your albums: " + err.message);
    } else {
      var albums = data.CommonPrefixes.map(function (commonPrefix) {
        var prefix = commonPrefix.Prefix;
        var albumName = decodeURIComponent(prefix.replace("/", ""));
```

```
        return getHtml([
          "<li>",
          "<span onclick=\"deleteAlbum('" + albumName + "')\">X</span>",
          "<span onclick=\"viewAlbum('" + albumName + "')\">",
          albumName,
          "</span>",
          "</li>",
        ]);
      });
      var message = albums.length
        ? getHtml([
            "<p>Click on an album name to view it.</p>",
            "<p>Click on the X to delete the album.</p>",
          ])
        : "<p>You do not have any albums. Please Create album.";
      var htmlTemplate = [
        "<h2>Albums</h2>",
        message,
        "<ul>",
        getHtml(albums),
        "</ul>",
        "<button onclick=\"createAlbum(prompt('Enter Album Name:'))\">",
        "Create New Album",
        "</button>",
      ];
      document.getElementById("app").innerHTML = getHtml(htmlTemplate);
    }
  });
}

function createAlbum(albumName) {
  albumName = albumName.trim();
  if (!albumName) {
    return alert("Album names must contain at least one non-space character.");
  }
  if (albumName.indexOf("/") !== -1) {
    return alert("Album names cannot contain slashes.");
  }
  var albumKey = encodeURIComponent(albumName);
  s3.headObject({ Key: albumKey }, function (err, data) {
    if (!err) {
      return alert("Album already exists.");
    }
    if (err.code !== "NotFound") {
```

```
      return alert("There was an error creating your album: " + err.message);
    }
    s3.putObject({ Key: albumKey }, function (err, data) {
      if (err) {
        return alert("There was an error creating your album: " + err.message);
      }
      alert("Successfully created album.");
      viewAlbum(albumName);
    });
  });
}

function viewAlbum(albumName) {
  var albumPhotosKey = encodeURIComponent(albumName) + "/";
  s3.listObjects({ Prefix: albumPhotosKey }, function (err, data) {
    if (err) {
      return alert("There was an error viewing your album: " + err.message);
    }
    // 'this' references the AWS.Response instance that represents the response
    var href = this.request.httpRequest.endpoint.href;
    var bucketUrl = href + albumBucketName + "/";

    var photos = data.Contents.map(function (photo) {
      var photoKey = photo.Key;
      var photoUrl = bucketUrl + encodeURIComponent(photoKey);
      return getHtml([
        "<span>",
        "<div>",
        '<img style="width:128px;height:128px;" src="' + photoUrl + '"/>',
        "</div>",
        "<div>",
        "<span onclick=\"deletePhoto('" +
          albumName +
          "','" +
          photoKey +
          "')\">",
        "X",
        "</span>",
        "<span>",
        photoKey.replace(albumPhotosKey, ""),
        "</span>",
        "</div>",
        "</span>",
      ]);
```

```
    });
    var message = photos.length
      ? "<p>Click on the X to delete the photo</p>"
      : "<p>You do not have any photos in this album. Please add photos.</p>";
    var htmlTemplate = [
      "<h2>",
      "Album: " + albumName,
      "</h2>",
      message,
      "<div>",
      getHtml(photos),
      "</div>",
      '<input id="photoupload" type="file" accept="image/*">',
      '<button id="addphoto" onclick="addPhoto(\'' + albumName + "')\">",
      "Add Photo",
      "</button>",
      '<button onclick="listAlbums()">',
      "Back To Albums",
      "</button>",
    ];
    document.getElementById("app").innerHTML = getHtml(htmlTemplate);
  });
}

function addPhoto(albumName) {
  var files = document.getElementById("photoupload").files;
  if (!files.length) {
    return alert("Please choose a file to upload first.");
  }
  var file = files[0];
  var fileName = file.name;
  var albumPhotosKey = encodeURIComponent(albumName) + "/";

  var photoKey = albumPhotosKey + fileName;

  // Use S3 ManagedUpload class as it supports multipart uploads
  var upload = new AWS.S3.ManagedUpload({
    params: {
      Bucket: albumBucketName,
      Key: photoKey,
      Body: file,
    },
  });
```

```
  var promise = upload.promise();

  promise.then(
    function (data) {
      alert("Successfully uploaded photo.");
      viewAlbum(albumName);
    },
    function (err) {
      return alert("There was an error uploading your photo: ", err.message);
    }
  );
}

function deletePhoto(albumName, photoKey) {
  s3.deleteObject({ Key: photoKey }, function (err, data) {
    if (err) {
      return alert("There was an error deleting your photo: ", err.message);
    }
    alert("Successfully deleted photo.");
    viewAlbum(albumName);
  });
}

function deleteAlbum(albumName) {
  var albumKey = encodeURIComponent(albumName) + "/";
  s3.listObjects({ Prefix: albumKey }, function (err, data) {
    if (err) {
      return alert("There was an error deleting your album: ", err.message);
    }
    var objects = data.Contents.map(function (object) {
      return { Key: object.Key };
    });
    s3.deleteObjects(
      {
        Delete: { Objects: objects, Quiet: true },
      },
      function (err, data) {
        if (err) {
          return alert("There was an error deleting your album: ", err.message);
        }
        alert("Successfully deleted album.");
        listAlbums();
      }
    );
```

```
  });
}
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

Amazon S3 Node.js 예제

다음 주제는 브라우저에서 Node.js를 사용하는 Amazon S3 버킷과의 상호 작용을 위해 AWS SDK for JavaScript가 사용되는 방식의 예제 두 가지를 설명합니다.

주제

- Amazon S3 버킷 생성 및 사용
- Amazon S3 버킷 구성
- Amazon S3 버킷 액세스 권한 관리
- Amazon S3 버킷 정책 작업
- Amazon S3 버킷을 정적 웹 호스트로 사용

Amazon S3 버킷 생성 및 사용

이 Node.js 코드 예제는 다음을 보여 줍니다.

- 계정에서 Amazon S3 버킷 목록을 가져오고 표시하는 방법
- Amazon S3 버킷을 생성하는 방법
- 지정된 버킷에 객체를 업로드하는 방법.

시나리오

이 예제에서는 일련의 Node.js 모듈을 사용하여 기존 Amazon S3 버킷 목록을 가져오고 버킷을 생성하며 지정된 버킷에 파일을 업로드합니다. Node.js 모듈은 다음 Amazon S3 클라이언트 클래스 메서드를 사용하여 Amazon S3 버킷에서 정보를 가져오고 파일을 업로드하기 위해 SDK for JavaScript를 사용합니다.

- `listBuckets`

- `createBucket`
- `listObjects`
- `upload`
- `deleteBucket`

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 이러한 작업들을 완료해야 합니다.

- Node.js를 설치합니다. Node.js 설치에 대한 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 공유 자격 증명 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.

SDK 구성

글로벌 구성 객체를 생성한 후 코드에 대한 리전을 설정하여 SDK for JavaScript를 구성합니다. 이 예제에서 리전이 `us-west-2`로 설정되어 있습니다.

```
// Load the SDK for JavaScript
var AWS = require('aws-sdk');
// Set the Region
AWS.config.update({region: 'us-west-2'});
```

Amazon S3 버킷 목록 표시

파일 이름이 `s3_listbuckets.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. Amazon Simple Storage Service에 액세스하려면 `AWS.S3` 서비스 객체를 생성합니다. Amazon S3 서비스 객체의 `listBuckets` 메서드를 직접 호출하여 버킷의 목록을 검색합니다. 콜백 함수의 `data` 파라미터에는 버킷을 표시하기 위한 맵 배열을 포함하는 `Buckets` 속성이 있습니다. 콘솔에 로그인하여 버킷 목록을 표시합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create S3 service object
s3 = new AWS.S3({ apiVersion: "2006-03-01" });
```

```
// Call S3 to list the buckets
s3.listBuckets(function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.Buckets);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node s3_listbuckets.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

Amazon S3 버킷 생성

파일 이름이 `s3_createbucket.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. `AWS.S3` 서비스 객체를 생성합니다. 이 모듈은 단일 명령줄 인수를 가져와서 새 버킷의 이름을 지정합니다.

새로 생성된 버킷의 이름을 포함하여 Amazon S3 서비스 객체의 `createBucket` 메서드를 직접 호출하는 데 사용되는 파라미터를 담을 변수를 추가합니다. Amazon S3가 버킷을 성공적으로 생성한 후 콜백 함수가 새 버킷의 위치를 콘솔에 로깅합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create S3 service object
s3 = new AWS.S3({ apiVersion: "2006-03-01" });

// Create the parameters for calling createBucket
var bucketParams = {
  Bucket: process.argv[2],
};

// call S3 to create the bucket
s3.createBucket(bucketParams, function (err, data) {
  if (err) {
```

```
    console.log("Error", err);
  } else {
    console.log("Success", data.Location);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node s3_createbucket.js BUCKET_NAME
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

Amazon S3 버킷에 파일 업로드

파일 이름이 `s3_upload.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. `AWS.S3` 서비스 객체를 생성합니다. 이 모듈은 두 개의 명령줄 인수를 가져옵니다. 첫 번째 인수는 대상 버킷을 지정하고 두 번째 인수는 업로드할 파일을 지정합니다.

Amazon S3 서비스 객체의 `upload` 메서드를 직접 호출하는 데 필요한 파라미터가 있는 변수를 생성합니다. 대상 버킷의 이름을 `Bucket` 파라미터에 제공합니다. `Key` 파라미터는 선택한 파일의 이름으로 설정되며, 이 이름은 Node.js `path` 모듈을 사용하여 가져올 수 있습니다. `Body` 파라미터는 파일의 콘텐츠로 설정되며, 이 콘텐츠는 Node.js `createReadStream` 모듈의 `fs`을 사용하여 가져올 수 있습니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create S3 service object
var s3 = new AWS.S3({ apiVersion: "2006-03-01" });

// call S3 to retrieve upload file to specified bucket
var uploadParams = { Bucket: process.argv[2], Key: "", Body: "" };
var file = process.argv[3];

// Configure the file stream and obtain the upload parameters
var fs = require("fs");
var fileStream = fs.createReadStream(file);
fileStream.on("error", function (err) {
  console.log("File Error", err);
});
```

```
uploadParams.Body = fileStream;
var path = require("path");
uploadParams.Key = path.basename(file);

// call S3 to retrieve upload file to specified bucket
s3.upload(uploadParams, function (err, data) {
  if (err) {
    console.log("Error", err);
  }
  if (data) {
    console.log("Upload Success", data.Location);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node s3_upload.js BUCKET_NAME FILE_NAME
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

Amazon S3 버킷의 객체 나열

파일 이름이 `s3_listobjects.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. `AWS.S3` 서비스 객체를 생성합니다.

읽을 버킷의 이름을 포함하여 Amazon S3 서비스 객체의 `listObjects` 메서드를 호출하는 데 사용되는 파라미터를 담을 변수를 추가합니다. 콜백 함수는 객체(파일)의 목록 또는 실패 메시지를 로깅합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create S3 service object
s3 = new AWS.S3({ apiVersion: "2006-03-01" });

// Create the parameters for calling listObjects
var bucketParams = {
  Bucket: "BUCKET_NAME",
};
```

```
// Call S3 to obtain a list of the objects in the bucket
s3.listObjects(bucketParams, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node s3_listobjects.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

Amazon S3 버킷 삭제

파일 이름이 `s3_deletebucket.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. `AWS.S3` 서비스 객체를 생성합니다.

삭제할 버킷의 이름을 포함하여 Amazon S3 서비스 객체의 `createBucket` 메서드를 호출하는 데 사용되는 파라미터를 담을 변수를 추가합니다. 버킷을 삭제하려면 버킷이 비어 있어야 합니다. 콜백 함수는 성공 또는 실패 메시지를 로깅합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create S3 service object
s3 = new AWS.S3({ apiVersion: "2006-03-01" });

// Create params for S3.deleteBucket
var bucketParams = {
  Bucket: "BUCKET_NAME",
};

// Call S3 to delete the bucket
s3.deleteBucket(bucketParams, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
```

```
    console.log("Success", data);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node s3_deletebucket.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

Amazon S3 버킷 구성

이 Node.js 코드 예제는 다음을 보여 줍니다.

- 버킷에 대한 CORS(교차 오리진 리소스 공유) 권한을 구성하는 방법.

시나리오

이 예제에서는 일련의 Node.js 모듈을 사용하여 Amazon S3 버킷 목록을 표시하고 CORS 및 버킷 로깅을 구성합니다. Node.js 모듈은 SDK for JavaScript로 Amazon S3 클라이언트 클래스의 다음 메서드를 사용하여 선택한 Amazon S3 버킷을 구성합니다.

- `getBucketCors`
- `putBucketCors`

Amazon S3 버킷을 사용하는 CORS 구성 사용에 대한 자세한 내용은 Amazon Simple Storage Service 사용 설명서의 CORS(Cross-Origin Resource Sharing)를 참조하세요.

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 이러한 작업들을 완료해야 합니다.

- Node.js를 설치합니다. Node.js 설치에 대한 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 공유 자격 증명 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.

SDK 구성

글로벌 구성 객체를 생성한 후 코드에 대한 리전을 설정하여 SDK for JavaScript를 구성합니다. 이 예제에서 리전이 `us-west-2`로 설정되어 있습니다.

```
// Load the SDK for JavaScript
var AWS = require('aws-sdk');
// Set the Region
AWS.config.update({region: 'us-west-2'});
```

버킷 CORS 구성 검색

파일 이름이 `s3_getcors.js`인 Node.js 모듈을 생성합니다. 이 모듈은 단일 명령줄 인수를 가져와서 원하는 CORS 구성이 있는 버킷을 지정합니다. 위와 같이 SDK를 구성해야 합니다. `AWS.S3` 서비스 객체를 생성합니다.

전달해야 하는 유일한 파라미터는 `getBucketCors` 메서드를 호출할 때 선택한 버킷의 이름입니다. 현재 버킷에 CORS 구성이 있는 경우 해당 구성은 Amazon S3에서 콜백 함수에 전달되는 `data` 파라미터의 `CORSRules` 속성으로 반환됩니다.

선택한 버킷에 CORS 구성이 없는 경우 해당 정보는 `error` 파라미터에서 콜백 함수에 반환됩니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create S3 service object
s3 = new AWS.S3({ apiVersion: "2006-03-01" });

// Set the parameters for S3.getBucketCors
var bucketParams = { Bucket: process.argv[2] };

// call S3 to retrieve CORS configuration for selected bucket
s3.getBucketCors(bucketParams, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else if (data) {
    console.log("Success", JSON.stringify(data.CORSRules));
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node s3_getcors.js BUCKET_NAME
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

버킷 CORS 구성 설정

파일 이름이 `s3_setcors.js`인 Node.js 모듈을 생성합니다. 이 모듈은 여러 명령줄 인수를 가져옵니다. 이러한 인수 중 첫 번째는 설정할 CORS 구성이 있는 버킷을 지정합니다. 추가 인수는 버킷에 허용할 HTTP 메서드(POST, GET, PUT, PATCH, DELETE, POST)를 열거합니다. 위와 같이 SDK를 구성합니다.

`AWS.S3` 서비스 객체를 생성합니다. 다음에는 `putBucketCors` 서비스 객체의 `AWS.S3` 메서드에 필요한 CORS 구성의 값을 담을 JSON 객체를 생성합니다. `"Authorization"` 값에 `AllowedHeaders`을 지정하고 `"*"` 값에 `AllowedOrigins`를 지정합니다. 처음에는 비어 있는 배열로 `AllowedMethods`의 값을 설정합니다.

허용된 메서드를 명령줄 파라미터로 Node.js 모듈에 지정하여 파라미터 중 하나와 일치하는 각 메서드를 추가합니다. 결과적으로 생성된 CORS 구성을 `CORSRules` 파라미터에 포함된 구성 배열에 추가합니다. `Bucket` 파라미터에서 CORS에 구성할 버킷을 지정합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create S3 service object
s3 = new AWS.S3({ apiVersion: "2006-03-01" });

// Create initial parameters JSON for putBucketCors
var thisConfig = {
  AllowedHeaders: ["Authorization"],
  AllowedMethods: [],
  AllowedOrigins: ["*"],
  ExposeHeaders: [],
  MaxAgeSeconds: 3000,
};

// Assemble the list of allowed methods based on command line parameters
var allowedMethods = [];
process.argv.forEach(function (val, index, array) {
```

```
  if (val.toUpperCase() === "POST") {
    allowedMethods.push("POST");
  }
  if (val.toUpperCase() === "GET") {
    allowedMethods.push("GET");
  }
  if (val.toUpperCase() === "PUT") {
    allowedMethods.push("PUT");
  }
  if (val.toUpperCase() === "PATCH") {
    allowedMethods.push("PATCH");
  }
  if (val.toUpperCase() === "DELETE") {
    allowedMethods.push("DELETE");
  }
  if (val.toUpperCase() === "HEAD") {
    allowedMethods.push("HEAD");
  }
});

// Copy the array of allowed methods into the config object
thisConfig.AllowedMethods = allowedMethods;
// Create array of configs then add the config object to it
var corsRules = new Array(thisConfig);

// Create CORS params
var corsParams = {
  Bucket: process.argv[2],
  CORSConfiguration: { CORSRules: corsRules },
};

// set the new CORS configuration on the selected bucket
s3.putBucketCors(corsParams, function (err, data) {
  if (err) {
    // display error message
    console.log("Error", err);
  } else {
    // update the displayed CORS config for the selected bucket
    console.log("Success", data);
  }
});
```

예제를 실행하려면 다음과 같이 하나 이상의 HTTP 메서드를 포함하여 명령줄에 다음을 입력합니다.

```
node s3_setcors.js BUCKET_NAME get put
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

Amazon S3 버킷 액세스 권한 관리

이 Node.js 코드 예제는 다음을 보여 줍니다.

- Amazon S3 버킷에 대한 액세스 제어 목록을 가져오거나 설정하는 방법.

시나리오

이 예제에서는 Node.js 모듈을 사용하여 선택한 버킷에 대한 버킷 ACL(액세스 제어 목록)을 표시하고 선택한 버킷에 대한 ACL에 변경 사항을 적용합니다. Node.js 모듈은 SDK for JavaScript로 Amazon S3 클라이언트 클래스의 다음 메서드를 사용하여 Amazon S3 버킷 액세스 권한을 관리합니다.

- `getBucketAcl`
- `putBucketAcl`

Amazon S3 버킷 액세스 제어 목록(ACL)에 대한 자세한 내용은 Amazon Simple Storage Service 사용 안내서의 ACL을 사용한 액세스 관리를 참조하세요.

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 이러한 작업들을 완료해야 합니다.

- Node.js를 설치합니다. Node.js 설치에 대한 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 공유 자격 증명 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.

SDK 구성

글로벌 구성 객체를 생성한 후 코드에 대한 리전을 설정하여 SDK for JavaScript를 구성합니다. 이 예제에서 리전이 `us-west-2`로 설정되어 있습니다.

```
// Load the SDK for JavaScript
var AWS = require('aws-sdk');
// Set the Region
AWS.config.update({region: 'us-west-2'});
```

현재 버킷 액세스 제어 목록 검색

파일 이름이 `s3_getbucketacl.js`인 Node.js 모듈을 생성합니다. 이 모듈은 단일 명령줄 인수를 가져와서 원하는 ACL 구성이 있는 버킷을 지정합니다. 위와 같이 SDK를 구성해야 합니다.

`AWS.S3` 서비스 객체를 생성합니다. 전달해야 하는 유일한 파라미터는 `getBucketAcl` 메서드를 호출할 때 선택한 버킷의 이름입니다. 현재 액세스 제어 목록(ACL) 구성은 Amazon S3가 콜백 함수에 전달하는 `data` 파라미터에서 반환됩니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create S3 service object
s3 = new AWS.S3({ apiVersion: "2006-03-01" });

var bucketParams = { Bucket: process.argv[2] };
// call S3 to retrieve policy for selected bucket
s3.getBucketAcl(bucketParams, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else if (data) {
    console.log("Success", data.Grants);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node s3_getbucketacl.js BUCKET_NAME
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

Amazon S3 버킷 정책 작업

이 Node.js 코드 예제는 다음을 보여 줍니다.

- Amazon S3 버킷의 버킷 정책을 검색하는 방법
- Amazon S3 버킷의 버킷 정책을 추가하거나 업데이트하는 방법
- Amazon S3 버킷의 버킷 정책을 삭제하는 방법

시나리오

이 예제에서는 일련의 Node.js 모듈을 사용하여 Amazon S3 버킷에 대한 버킷 정책을 검색, 설정 또는 삭제합니다. Node.js 모듈은 SDK for JavaScript로 Amazon S3 클라이언트 클래스의 다음 메서드를 사용하여 선택한 Amazon S3 버킷에 대한 정책을 구성합니다.

- `getBucketPolicy`
- `putBucketPolicy`
- `deleteBucketPolicy`

Amazon S3의 버킷 정책에 대한 자세한 내용은 Amazon Simple Storage Service 사용 설명서의 버킷 정책 및 사용자 정책 사용을 참조하세요.

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 이러한 작업들을 완료해야 합니다.

- Node.js를 설치합니다. Node.js 설치에 대한 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 공유 자격 증명 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.

SDK 구성

글로벌 구성 객체를 생성한 후 코드에 대한 리전을 설정하여 SDK for JavaScript를 구성합니다. 이 예제에서 리전이 `us-west-2`로 설정되어 있습니다.

```
// Load the SDK for JavaScript
var AWS = require('aws-sdk');
// Set the Region
AWS.config.update({region: 'us-west-2'});
```

현재 버킷 정책 검색

파일 이름이 `s3_getbucketpolicy.js`인 Node.js 모듈을 생성합니다. 이 모듈은 원하는 정책이 있는 버킷을 지정하는 단일 명령줄 인수를 가져옵니다. 위와 같이 SDK를 구성해야 합니다.

`AWS.S3` 서비스 객체를 생성합니다. 전달해야 하는 유일한 파라미터는 `getBucketPolicy` 메서드를 호출할 때 선택한 버킷의 이름입니다. 현재 버킷에 정책이 있는 경우 해당 정책은 Amazon S3에서 콜백 함수에 전달되는 `data` 파라미터에 반환됩니다.

선택한 버킷에 정책이 없는 경우 해당 정보는 `error` 파라미터에서 콜백 함수에 반환됩니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create S3 service object
s3 = new AWS.S3({ apiVersion: "2006-03-01" });

var bucketParams = { Bucket: process.argv[2] };
// call S3 to retrieve policy for selected bucket
s3.getBucketPolicy(bucketParams, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else if (data) {
    console.log("Success", data.Policy);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node s3_getbucketpolicy.js BUCKET_NAME
```

이 샘플 코드는 [GitHub](https://github.com)에서 찾을 수 있습니다.

간단한 버킷 정책 설정

파일 이름이 `s3_setbucketpolicy.js`인 Node.js 모듈을 생성합니다. 이 모듈은 적용할 정책이 있는 버킷을 지정하는 단일 명령줄 인수를 가져옵니다. 위와 같이 SDK를 구성합니다.

`AWS.S3` 서비스 객체를 생성합니다. 버킷 정책은 JSON 문서에서 지정됩니다. 먼저, 버킷을 식별하는 `Resource` 값을 제외하고 정책을 지정하기 위한 모든 값을 포함하는 JSON 객체를 생성합니다.

정책에 필요한 `Resource` 문자열의 형식을 지정하여 선택한 버킷의 이름을 통합합니다. 해당 문자열을 JSON 객체에 삽입합니다. 버킷의 이름과 문자열 값으로 변환된 JSON 정책을 포함하여 `putBucketPolicy` 메서드를 위한 파라미터를 준비합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create S3 service object
s3 = new AWS.S3({ apiVersion: "2006-03-01" });

var readOnlyAnonUserPolicy = {
  Version: "2012-10-17",
  Statement: [
    {
      Sid: "AddPerm",
      Effect: "Allow",
      Principal: "*",
      Action: ["s3:GetObject"],
      Resource: [""],
    },
  ],
};

// create selected bucket resource string for bucket policy
var bucketResource = "arn:aws:s3:::" + process.argv[2] + "/*";
readOnlyAnonUserPolicy.Statement[0].Resource[0] = bucketResource;

// convert policy JSON into string and assign into params
var bucketPolicyParams = {
  Bucket: process.argv[2],
  Policy: JSON.stringify(readOnlyAnonUserPolicy),
};
```

```
// set the new policy on the selected bucket
s3.putBucketPolicy(bucketPolicyParams, function (err, data) {
  if (err) {
    // display error message
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node s3_setbucketpolicy.js BUCKET_NAME
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

버킷 정책 삭제

파일 이름이 `s3_deletebucketpolicy.js`인 Node.js 모듈을 생성합니다. 이 모듈은 삭제할 정책이 있는 버킷을 지정하는 단일 명령줄 인수를 가져옵니다. 위와 같이 SDK를 구성합니다.

`AWS.S3` 서비스 객체를 생성합니다. `deleteBucketPolicy` 메서드를 호출할 때 전달해야 하는 유일한 파라미터는 선택한 버킷의 이름입니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create S3 service object
s3 = new AWS.S3({ apiVersion: "2006-03-01" });

var bucketParams = { Bucket: process.argv[2] };
// call S3 to delete policy for selected bucket
s3.deleteBucketPolicy(bucketParams, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else if (data) {
    console.log("Success", data);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node s3_deletebucketpolicy.js BUCKET_NAME
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

Amazon S3 버킷을 정적 웹 호스트로 사용

이 Node.js 코드 예제는 다음을 보여 줍니다.

- Amazon S3 버킷을 정적 웹 호스트로 설정하는 방법

시나리오

이 예제에서는 일련의 Node.js 모듈을 사용하여 정적 웹 호스트 역할을 하는 버킷 중 하나를 구성합니다. Node.js 모듈은 SDK for JavaScript로 Amazon S3 클라이언트 클래스의 다음 메서드를 사용하여 선택한 Amazon S3 버킷을 구성합니다.

- `getBucketWebsite`
- `putBucketWebsite`
- `deleteBucketWebsite`

Amazon S3 버킷을 정적 웹 호스트로 사용하는 방법에 대한 자세한 내용은 Amazon Simple Storage Service 사용 설명서의 Amazon S3에서 정적 웹 사이트 호스팅을 참조하세요.

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 이러한 작업들을 완료해야 합니다.

- Node.js를 설치합니다. Node.js 설치에 대한 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 공유 자격 증명 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.

SDK 구성

글로벌 구성 객체를 생성한 후 코드에 대한 리전을 설정하여 SDK for JavaScript를 구성합니다. 이 예제에서 리전이 `us-west-2`로 설정되어 있습니다.

```
// Load the SDK for JavaScript
var AWS = require('aws-sdk');
// Set the Region
AWS.config.update({region: 'us-west-2'});
```

현재 버킷 웹 사이트 구성 검색

파일 이름이 `s3_getbucketwebsite.js`인 Node.js 모듈을 생성합니다. 이 모듈은 원하는 웹 사이트 구성이 있는 버킷을 지정하는 단일 명령줄 인수를 가져옵니다. 위와 같이 SDK를 구성합니다.

`AWS.S3` 서비스 객체를 생성합니다. 버킷 목록에서 선택한 버킷에 대한 현재 버킷 웹 사이트 구성을 검색하는 함수를 생성합니다. 전달해야 하는 유일한 파라미터는 `getBucketWebsite` 메서드를 호출할 때 선택한 버킷의 이름입니다. 현재 버킷에 웹 사이트가 구성된 경우 해당 구성은 Amazon S3에서 콜백 함수에 전달되는 `data` 파라미터에 반환됩니다.

선택한 버킷에 웹 사이트 구성이 없는 경우 해당 정보는 `err` 파라미터에서 콜백 함수에 반환됩니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create S3 service object
s3 = new AWS.S3({ apiVersion: "2006-03-01" });

var bucketParams = { Bucket: process.argv[2] };

// call S3 to retrieve the website configuration for selected bucket
s3.getBucketWebsite(bucketParams, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else if (data) {
    console.log("Success", data);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node s3_getbucketwebsite.js BUCKET_NAME
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

버킷 웹 사이트 구성 설정

파일 이름이 `s3_setbucketwebsite.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. `AWS.S3` 서비스 객체를 생성합니다.

버킷 웹 사이트 구성을 적용하는 함수를 생성합니다. 선택한 버킷은 이 구성을 사용하여 정적 웹 호스트의 역할을 수행할 수 있습니다. 웹 사이트 구성은 JSON에서 지정됩니다. 먼저, 오류 문서를 식별하는 `Key` 값과 인덱스 문서를 식별하는 `Suffix` 값을 제외하고 웹 사이트 구성을 지정하기 위한 모든 값을 포함하는 JSON 객체를 생성합니다.

텍스트 입력 요소의 값을 JSON 객체에 삽입합니다. 버킷의 이름과 JSON 웹 사이트 구성을 포함하여 `putBucketWebsite` 메서드를 위한 파라미터를 준비합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create S3 service object
s3 = new AWS.S3({ apiVersion: "2006-03-01" });

// Create JSON for putBucketWebsite parameters
var staticHostParams = {
  Bucket: "",
  WebsiteConfiguration: {
    ErrorDocument: {
      Key: "",
    },
    IndexDocument: {
      Suffix: "",
    },
  },
};

// Insert specified bucket name and index and error documents into params JSON
// from command line arguments
staticHostParams.Bucket = process.argv[2];
staticHostParams.WebsiteConfiguration.IndexDocument.Suffix = process.argv[3];
```

```
staticHostParams.WebsiteConfiguration.ErrorDocument.Key = process.argv[4];

// set the new website configuration on the selected bucket
s3.putBucketWebsite(staticHostParams, function (err, data) {
  if (err) {
    // display error message
    console.log("Error", err);
  } else {
    // update the displayed website configuration for the selected bucket
    console.log("Success", data);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node s3_setbucketwebsite.js BUCKET_NAME INDEX_PAGE ERROR_PAGE
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

버킷 웹 사이트 구성 삭제

파일 이름이 `s3_deletebucketwebsite.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. `AWS.S3` 서비스 객체를 생성합니다.

선택한 버킷에 대한 웹 사이트 구성을 삭제하는 함수를 생성합니다. `deleteBucketWebsite` 메서드를 호출할 때 전달해야 하는 유일한 파라미터는 선택한 버킷의 이름입니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create S3 service object
s3 = new AWS.S3({ apiVersion: "2006-03-01" });

var bucketParams = { Bucket: process.argv[2] };

// call S3 to delete website configuration for selected bucket
s3.deleteBucketWebsite(bucketParams, function (error, data) {
  if (error) {
    console.log("Error", err);
  } else if (data) {
```

```
    console.log("Success", data);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node s3_deletebucketwebsite.js BUCKET_NAME
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

Amazon Simple Email Services 예제

Amazon Simple Email Service(Amazon SES)는 디지털 마케팅 담당자 및 애플리케이션 개발자가 마케팅, 알림 및 거래 이메일을 전송하는 데 도움이 되도록 설계된 클라우드 기반 이메일 전송 서비스입니다. 이 서비스는 이메일을 사용하여 고객과 연락하는 모든 규모의 기업을 위한 안정적이고 비용 효과적인 서비스입니다.

Amazon SES용 JavaScript API는 `AWS.SES` 클라이언트 클래스를 통해 노출됩니다. Amazon SES 클라이언트 클래스 사용에 대한 자세한 내용은 API 참조의 `Class: AWS.SES` 섹션을 참조하세요.

주제

- Amazon SES 자격 증명 관리
- Amazon SES에서 이메일 템플릿 작업
- Amazon SES로 이메일 전송
- Amazon SES에서 이메일 수신을 위한 IP 주소 필터 사용
- Amazon SES에서 수신 규칙 사용

Amazon SES 자격 증명 관리

이 Node.js 코드 예제는 다음을 보여 줍니다.

- Amazon SES에 사용되는 이메일 주소 및 도메인을 확인하는 방법
- IAM 정책을 Amazon SES 자격 증명에 할당하는 방법
- AWS 계정의 모든 Amazon SES 자격 증명을 나열하는 방법
- Amazon SES에 사용되는 자격 증명을 삭제하는 방법

Amazon SES 자격 증명은 Amazon SES에서 이메일을 보내는 데 사용하는 이메일 주소 또는 도메인입니다. Amazon SES에서는 이메일 자격 증명을 확인해야 합니다. 이렇게 해당 자격 증명을 소유하고 있음을 확인하고 다른 사람이 이를 사용하지 못하게 방지합니다.

Amazon SES에서 이메일 주소 및 도메인을 확인하는 방법에 대한 자세한 내용은 Amazon Simple Email Service 개발자 안내서의 이메일 주소 및 도메인 확인 섹션을 참조하세요. Amazon SES의 전송 권한 부여에 관한 자세한 내용은 Amazon SES 전송 권한 부여의 개요 섹션을 참조하세요.

시나리오

이 예에서는 일련의 Node.js 모듈을 사용하여 Amazon SES 자격 증명을 확인하고 관리합니다. 이 Node.js 모듈은 SDK for JavaScript에서 `AWS.SES` 클라이언트 클래스의 다음 메서드를 사용하여 이메일 주소와 도메인을 확인합니다.

- `listIdentities`
- `deleteIdentity`
- `verifyEmailIdentity`
- `verifyDomainIdentity`

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 이러한 작업들을 완료해야 합니다.

- Node.js를 설치합니다. Node.js 설치에 대한 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 자격 증명 JSON 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.

SDK 구성

글로벌 구성 객체를 생성한 후 코드에 대한 리전을 설정하여 SDK for JavaScript를 구성합니다. 이 예제에서 리전이 us-west-2로 설정되어 있습니다.

```
// Load the SDK for JavaScript
var AWS = require('aws-sdk');

// Set the Region
AWS.config.update({region: 'us-west-2'});
```

자격 증명 나열

이 예에서는 Node.js 모듈을 사용하여 Amazon SES에 사용할 이메일 주소와 도메인을 나열합니다. 파일 이름이 ses_listidentities.js인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성합니다.

AWS.SES 클라이언트 클래스의 listIdentities 메서드에 대한 IdentityType 및 기타 파라미터를 전달할 객체를 생성합니다. listIdentities 메서드를 직접 호출하려면 파라미터 객체를 전달하는 Amazon SES 서비스 객체를 간접 호출하기 위한 promise를 생성합니다.

그런 다음 promise 콜백에서 response를 처리합니다. promise에서 반환된 data는 IdentityType 파라미터에 지정된 도메인 자격 증명의 배열을 포함하고 있습니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set region
AWS.config.update({ region: "REGION" });

// Create listIdentities params
var params = {
  IdentityType: "Domain",
  MaxItems: 10,
};

// Create the promise and SES service object
var listIDsPromise = new AWS.SES({ apiVersion: "2010-12-01" })
```

```
  .listIdentities(params)
  .promise();

// Handle promise's fulfilled/rejected states
listIDsPromise
  .then(function (data) {
    console.log(data.Identities);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node ses_listidentities.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

이메일 주소 자격 증명 확인

이 예에서는 Node.js 모듈을 사용하여 Amazon SES에 사용할 이메일 발신자를 확인합니다. 파일 이름이 `ses_verifyemailidentity.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성합니다. Amazon EC2에 액세스하려면 `AWS.SES` 서비스 객체를 생성합니다.

`AWS.SES` 클라이언트 클래스의 `verifyEmailIdentity` 메서드에 대한 `EmailAddress` 파라미터를 전달할 객체를 생성합니다. `verifyEmailIdentity` 메서드를 호출하려면 파라미터를 전달하는 Amazon SES 서비스 객체를 호출하기 위한 promise를 생성합니다. 그런 다음 promise 콜백에서 `response`를 처리합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set region
AWS.config.update({ region: "REGION" });

// Create promise and SES service object
var verifyEmailPromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .verifyEmailIdentity({ EmailAddress: "ADDRESS@DOMAIN.EXT" })
  .promise();

// Handle promise's fulfilled/rejected states
verifyEmailPromise
```

```
  .then(function (data) {
    console.log("Email verification initiated");
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

예제를 실행하려면 명령줄에서 다음을 입력합니다. 확인을 위해 Amazon SES에 도메인이 추가됩니다.

```
node ses_verifyemailidentity.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

도메인 자격 증명 확인

이 예에서는 Node.js 모듈을 사용하여 Amazon SES에 사용할 이메일 도메인을 확인합니다. 파일 이름이 `ses_verifydomainidentity.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성합니다.

`AWS.SES` 클라이언트 클래스의 `verifyDomainIdentity` 메서드에 대한 `Domain` 파라미터를 전달할 객체를 생성합니다. `verifyDomainIdentity` 메서드를 직접 호출하려면 파라미터 객체를 전달하는 Amazon SES 서비스 객체를 간접 호출하기 위한 promise를 생성합니다. 그런 다음 promise 콜백에서 `response`를 처리합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set region
AWS.config.update({ region: "REGION" });

// Create the promise and SES service object
var verifyDomainPromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .verifyDomainIdentity({ Domain: "DOMAIN_NAME" })
  .promise();

// Handle promise's fulfilled/rejected states
verifyDomainPromise
  .then(function (data) {
    console.log("Verification Token: " + data.VerificationToken);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

예제를 실행하려면 명령줄에서 다음을 입력합니다. 확인을 위해 Amazon SES에 도메인이 추가됩니다.

```
node ses_verifydomainidentity.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

자격 증명 삭제

이 예에서는 Node.js 모듈을 사용하여 Amazon SES에 사용되는 이메일 주소 또는 도메인을 삭제합니다. 파일 이름이 `ses_deleteidentity.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성합니다.

`AWS.SES` 클라이언트 클래스의 `deleteIdentity` 메서드에 대한 `Identity` 파라미터를 전달할 객체를 생성합니다. `deleteIdentity` 메서드를 직접 호출하려면 Amazon SES 서비스 객체를 간접 호출하기 위한 `request`를 생성하여 파라미터를 전달합니다. 그런 다음 promise 콜백에서 `response`를 처리합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set region
AWS.config.update({ region: "REGION" });

// Create the promise and SES service object
var deletePromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .deleteIdentity({ Identity: "DOMAIN_NAME" })
  .promise();

// Handle promise's fulfilled/rejected states
deletePromise
  .then(function (data) {
    console.log("Identity Deleted");
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node ses_deleteidentity.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

Amazon SES에서 이메일 템플릿 작업

이 Node.js 코드 예제는 다음을 보여 줍니다.

- 모든 이메일 템플릿의 목록을 가져옵니다.
- 이메일 템플릿을 검색하고 업데이트합니다.
- 이메일 템플릿을 생성하고 삭제합니다.

Amazon SES에서 이메일 템플릿을 사용하여 맞춤형 이메일 메시지를 전송할 수 있습니다. Amazon Simple Email Service(Amazon SES)에서 이메일 템플릿을 생성하고 사용하는 방법에 대한 자세한 내용은 Amazon Simple Email Service 개발자 안내서의 템플릿을 사용하여 Amazon SES API를 통해 맞춤형 이메일 전송 섹션을 참조하세요.

시나리오

이 예제에서는 일련의 Node.js 모듈을 사용하여 이메일 템플릿을 작업합니다. 이 Node.js 모듈은 SDK for JavaScript에서 `AWS.SES` 클라이언트 클래스의 다음 메서드를 사용하여 이메일 템플릿을 생성하고 사용합니다.

- `listTemplates`
- `createTemplate`
- `getTemplate`
- `deleteTemplate`
- `updateTemplate`

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 이러한 작업들을 완료해야 합니다.

- Node.js를 설치합니다. Node.js 설치에 대한 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 자격 증명 파일 생성에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.

이메일 템플릿 나열

이 예에서는 Node.js 모듈을 사용하여 Amazon SES에 사용할 이메일 템플릿을 생성합니다. 파일 이름이 `ses_listtemplates.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성합니다.

`AWS.SES` 클라이언트 클래스의 `listTemplates` 메서드에 대한 파라미터를 전달할 객체를 생성합니다. `listTemplates` 메서드를 호출하려면 파라미터를 전달하는 Amazon SES 서비스 객체를 호출하기 위한 promise를 생성합니다. 그런 다음 promise 콜백에서 `response`를 처리합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the promise and SES service object
var templatePromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .listTemplates({ MaxItems: ITEMS_COUNT })
  .promise();

// Handle promise's fulfilled/rejected states
templatePromise
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

예제를 실행하려면 명령줄에서 다음을 입력합니다. Amazon SES가 템플릿 목록을 반환합니다.

```
node ses_listtemplates.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

이메일 템플릿 가져오기

이 예에서는 Node.js 모듈을 사용하여 Amazon SES에 사용할 이메일 템플릿을 가져옵니다. 파일 이름이 `ses_gettemplate.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성합니다.

`AWS.SES` 클라이언트 클래스의 `getTemplate` 메서드에 대한 `TemplateName` 파라미터를 전달할 객체를 생성합니다. `getTemplate` 메서드를 호출하려면 파라미터를 전달하는 Amazon SES 서비스 객체를 호출하기 위한 promise를 생성합니다. 그런 다음 promise 콜백에서 `response`를 처리합니다.

```
// Load the AWS SDK for Node.js.
var AWS = require("aws-sdk");
// Set the AWS Region.
AWS.config.update({ region: "REGION" });

// Create the promise and Amazon Simple Email Service (Amazon SES) service object.
var templatePromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .getTemplate({ TemplateName: "TEMPLATE_NAME" })
  .promise();

// Handle promise's fulfilled/rejected states
templatePromise
  .then(function (data) {
    console.log(data.Template.SubjectPart);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

예제를 실행하려면 명령줄에서 다음을 입력합니다. Amazon SES가 템플릿 세부 정보를 반환합니다.

```
node ses_gettemplate.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

이메일 템플릿 생성

이 예에서는 Node.js 모듈을 사용하여 Amazon SES에 사용할 이메일 템플릿을 생성합니다. 파일 이름이 `ses_createtemplate.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성합니다.

`TemplateName`, `HtmlPart`, `SubjectPart` 및 `TextPart`를 포함하여 `AWS.SES` 클라이언트 클래스의 `createTemplate` 메서드에 대한 파라미터를 전달할 객체를 생성합니다. `createTemplate` 메서드를 호출하려면 파라미터를 전달하는 Amazon SES 서비스 객체를 호출하기 위한 promise를 생성합니다. 그런 다음 promise 콜백에서 `response`를 처리합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create createTemplate params
var params = {
```

```
  Template: {
    TemplateName: "TEMPLATE_NAME" /* required */,
    HtmlPart: "HTML_CONTENT",
    SubjectPart: "SUBJECT_LINE",
    TextPart: "TEXT_CONTENT",
  },
};

// Create the promise and SES service object
var templatePromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .createTemplate(params)
  .promise();

// Handle promise's fulfilled/rejected states
templatePromise
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

예제를 실행하려면 명령줄에서 다음을 입력합니다. 템플릿이 Amazon SES에 추가됩니다.

```
node ses_createtemplate.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

이메일 템플릿 업데이트

이 예에서는 Node.js 모듈을 사용하여 Amazon SES에 사용할 이메일 템플릿을 생성합니다. 파일 이름이 `ses_updatetemplate.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성합니다.

`AWS.SES` 클라이언트 클래스의 `updateTemplate` 메서드에 전달된 필수 `TemplateName` 파라미터와 함께 템플릿에서 업데이트하려는 `Template` 파라미터 값을 전달할 객체를 생성합니다. `updateTemplate` 메서드를 호출하려면 파라미터를 전달하는 Amazon SES 서비스 객체를 호출하기 위한 promise를 생성합니다. 그런 다음 promise 콜백에서 `response`를 처리합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });
```

```
// Create updateTemplate parameters
var params = {
  Template: {
    TemplateName: "TEMPLATE_NAME" /* required */,
    HtmlPart: "HTML_CONTENT",
    SubjectPart: "SUBJECT_LINE",
    TextPart: "TEXT_CONTENT",
  },
};

// Create the promise and SES service object
var templatePromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .updateTemplate(params)
  .promise();

// Handle promise's fulfilled/rejected states
templatePromise
  .then(function (data) {
    console.log("Template Updated");
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

예제를 실행하려면 명령줄에서 다음을 입력합니다. Amazon SES가 템플릿 세부 정보를 반환합니다.

```
node ses_updatetemplate.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

이메일 템플릿 삭제

이 예에서는 Node.js 모듈을 사용하여 Amazon SES에 사용할 이메일 템플릿을 생성합니다. 파일 이름이 `ses_deletetemplate.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성합니다.

`AWS.SES` 클라이언트 클래스의 `deleteTemplate` 메서드에 필수 `TemplateName` 파라미터를 전달할 객체를 생성합니다. `deleteTemplate` 메서드를 호출하려면 파라미터를 전달하는 Amazon SES 서비스 객체를 호출하기 위한 promise를 생성합니다. 그런 다음 promise 콜백에서 `response`를 처리합니다.

```
// Load the AWS SDK for Node.js
```

```
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the promise and SES service object
var templatePromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .deleteTemplate({ TemplateName: "TEMPLATE_NAME" })
  .promise();

// Handle promise's fulfilled/rejected states
templatePromise
  .then(function (data) {
    console.log("Template Deleted");
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

예제를 실행하려면 명령줄에서 다음을 입력합니다. Amazon SES가 템플릿 세부 정보를 반환합니다.

```
node ses_deletetemplate.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

Amazon SES로 이메일 전송

이 Node.js 코드 예제는 다음을 보여 줍니다.

- 테스트 또는 HTML 이메일을 전송합니다.
- 이메일 템플릿을 기반으로 이메일을 전송합니다.
- 이메일 템플릿을 기반으로 대량 이메일을 전송합니다.

Amazon SES API에서는 이메일 메시지 작성에 대해 원하는 제어 정도에 따라 서식 지정 및 원시라는 두 가지 이메일 전송 방법을 선택할 수 있습니다. 자세한 내용은 Amazon SES API를 사용하여 서식이 지정된 이메일 전송 및 Amazon SES API를 사용하여 원시 이메일 전송 섹션을 참조하세요.

시나리오

이 예제에서는 일련의 Node.js 모듈을 사용하여 다양한 방법으로 이메일을 전송합니다. 이 Node.js 모듈은 SDK for JavaScript에서 `AWS.SES` 클라이언트 클래스의 다음 메서드를 사용하여 이메일 템플릿을 생성하고 사용합니다.

- `sendEmail`
- `sendTemplatedEmail`
- `sendBulkTemplatedEmail`

사전 필수 작업

- Node.js를 설치합니다. Node.js 설치에 대한 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 자격 증명 JSON 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.

이메일 메시지 전송 요구 사항

Amazon SES에서는 이메일 메시지를 작성하는 즉시 전송 대기열에 넣습니다. `SES.sendEmail` 메서드를 사용하여 이메일을 전송하려면 메시지는 다음 요구 사항을 충족해야 합니다.

- 확인된 이메일 주소 또는 도메인에서 메시지를 전송해야 합니다. 확인되지 않은 주소 또는 도메인을 사용하여 이메일을 전송하려고 시도하면 작업 결과로 `"Email address not verified"` 오류가 발생합니다.
- 계정이 여전히 Amazon SES 샌드박스에 있는 경우 확인된 주소 또는 도메인으로만 또는 Amazon SES 메일박스 시뮬레이터와 연결된 이메일 주소로만 전송할 수 있습니다. 자세한 내용은 Amazon Simple Email Service 개발자 안내서의 이메일 주소 및 도메인 확인 섹션을 참조하세요.
- 첨부 파일을 포함한 메시지의 총 크기는 10MB 미만이어야 합니다.
- 메시지에 최소 하나 이상의 수신자 이메일 주소가 포함되어야 합니다. 수신자 주소는 받는 사람: 주소, 참조: 주소 또는 숨은 참조: 주소일 수 있습니다. 수신자 이메일 주소가 잘못된 경우(즉, `UserName@[SubDomain.]Domain.TopLevelDomain` 형식이 아닌 경우) 올바른 다른 수신자가 메시지가 포함되더라도 전체 메시지가 거부됩니다.
- 메시지에는 받는 사람:, 참조: 및 숨은 참조: 필드 전체에서 50명을 초과하는 수신자가 포함될 수 없습니다. 더 많은 대상에게 이메일 메시지를 전송해야 하는 경우 수신자 목록을 50명 이하의 여러 그룹으로 나눈 다음 `sendEmail` 메서드를 여러 번 호출하여 각 그룹에게 메시지를 전송할 수 있습니다.

이메일 전송

이 예제에서는 Node.js 모듈을 사용하여 Amazon SES에서 이메일을 전송합니다. 파일 이름이 `ses_sendemail.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성합니다.

발신자 및 수신자 주소, 제목, 일반 텍스트 및 HTML 형식의 이메일 본문을 포함하여 전송할 이메일을 정의하는 파라미터 값을 `AWS.SES` 클라이언트 클래스의 `sendEmail` 메서드에 전달할 객체를 생성합니다. `sendEmail` 메서드를 호출하려면 파라미터를 전달하는 Amazon SES 서비스 객체를 호출하기 위한 promise를 생성합니다. 그런 다음 promise 콜백에서 `response`를 처리합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create sendEmail params
var params = {
  Destination: {
    /* required */
    CcAddresses: [
      "EMAIL_ADDRESS",
      /* more items */
    ],
    ToAddresses: [
      "EMAIL_ADDRESS",
      /* more items */
    ],
  },
  Message: {
    /* required */
    Body: {
      /* required */
      Html: {
        Charset: "UTF-8",
        Data: "HTML_FORMAT_BODY",
      },
      Text: {
        Charset: "UTF-8",
        Data: "TEXT_FORMAT_BODY",
      },
    },
    Subject: {
      Charset: "UTF-8",
```

```
      Data: "Test email",
    },
  },
  Source: "SENDER_EMAIL_ADDRESS" /* required */,
  ReplyToAddresses: [
    "EMAIL_ADDRESS",
    /* more items */
  ],
};

// Create the promise and SES service object
var sendPromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .sendEmail(params)
  .promise();

// Handle promise's fulfilled/rejected states
sendPromise
  .then(function (data) {
    console.log(data.MessageId);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

예제를 실행하려면 명령줄에서 다음을 입력합니다. Amazon SES에서 전송할 이메일이 대기 상태로 전환됩니다.

```
node ses_sendemail.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

템플릿을 사용한 이메일 전송

이 예제에서는 Node.js 모듈을 사용하여 Amazon SES에서 이메일을 전송합니다. 파일 이름이 `ses_sendtemplatedemail.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성합니다.

발신자 및 수신자 주소, 제목, 일반 텍스트 및 HTML 형식의 이메일 본문을 포함하여 전송할 이메일을 정의하는 파라미터 값을 `AWS.SES` 클라이언트 클래스의 `sendTemplatedEmail` 메서드에 전달할 객체를 생성합니다. `sendTemplatedEmail` 메서드를 호출하려면 파라미터를 전달하는 Amazon SES 서비스 객체를 호출하기 위한 promise를 생성합니다. 그런 다음 promise 콜백에서 `response`를 처리합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create sendTemplatedEmail params
var params = {
  Destination: {
    /* required */
    CcAddresses: [
      "EMAIL_ADDRESS",
      /* more CC email addresses */
    ],
    ToAddresses: [
      "EMAIL_ADDRESS",
      /* more To email addresses */
    ],
  },
  Source: "EMAIL_ADDRESS" /* required */,
  Template: "TEMPLATE_NAME" /* required */,
  TemplateData: '{ "REPLACEMENT_TAG_NAME":"REPLACEMENT_VALUE" }' /* required */,
  ReplyToAddresses: ["EMAIL_ADDRESS"],
};

// Create the promise and SES service object
var sendPromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .sendTemplatedEmail(params)
  .promise();

// Handle promise's fulfilled/rejected states
sendPromise
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

예제를 실행하려면 명령줄에서 다음을 입력합니다. Amazon SES에서 전송할 이메일이 대기 상태로 전환됩니다.

```
node ses_sendtemplatedemail.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

템플릿을 사용한 대량 이메일 전송

이 예제에서는 Node.js 모듈을 사용하여 Amazon SES에서 이메일을 전송합니다. 파일 이름이 `ses_sendbulktemplatedemail.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성합니다.

발신자 및 수신자 주소, 제목, 일반 텍스트 및 HTML 형식의 이메일 본문을 포함하여 전송할 이메일을 정의하는 파라미터 값을 `AWS.SES` 클라이언트 클래스의 `sendBulkTemplatedEmail` 메서드에 전달할 객체를 생성합니다. `sendBulkTemplatedEmail` 메서드를 호출하려면 파라미터를 전달하는 Amazon SES 서비스 객체를 호출하기 위한 promise를 생성합니다. 그런 다음 promise 콜백에서 `response`를 처리합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create sendBulkTemplatedEmail params
var params = {
  Destinations: [
    /* required */
    {
      Destination: {
        /* required */
        CcAddresses: [
          "EMAIL_ADDRESS",
          /* more items */
        ],
        ToAddresses: [
          "EMAIL_ADDRESS",
          "EMAIL_ADDRESS",
          /* more items */
        ],
      },
      ReplacementTemplateData: '{ "REPLACEMENT_TAG_NAME":"REPLACEMENT_VALUE" }',
    },
  ],
  Source: "EMAIL_ADDRESS" /* required */,
  Template: "TEMPLATE_NAME" /* required */,
  DefaultTemplateData: '{ "REPLACEMENT_TAG_NAME":"REPLACEMENT_VALUE" }',
  ReplyToAddresses: ["EMAIL_ADDRESS"],
};
```

```
// Create the promise and SES service object
var sendPromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .sendBulkTemplatedEmail(params)
  .promise();

// Handle promise's fulfilled/rejected states
sendPromise
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.log(err, err.stack);
  });
```

예제를 실행하려면 명령줄에서 다음을 입력합니다. Amazon SES에서 전송할 이메일이 대기 상태로 전환됩니다.

```
node ses_sendbulktemplatedemail.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

Amazon SES에서 이메일 수신을 위한 IP 주소 필터 사용

이 Node.js 코드 예제는 다음을 보여 줍니다.

- 특정 IP 주소 또는 특정 범위의 IP 주소에서 발신한 메일을 수락하거나 거부하기 위한 IP 주소 필터를 생성합니다.
- 현재 IP 주소 필터를 나열합니다.
- IP 주소 필터를 삭제합니다.

Amazon SES에서 필터는 이름, IP 주소 범위, 해당 범위의 메일 허용 또는 차단 여부로 구성되는 데이터 구조입니다. 차단하거나 허용하려는 IP 주소는 CIDR(클래스 없는 도메인 간 라우팅) 표기법으로 단일 IP 주소 또는 IP 주소 범위로 지정됩니다. Amazon SES가 이메일을 수신하는 방법에 대한 자세한 내

용은 Amazon Simple Email Service 개발자 안내서의 Amazon SES 이메일 수신 개념 섹션을 참조하세요.

시나리오

이 예제에서는 일련의 Node.js 모듈을 사용하여 다양한 방법으로 이메일을 전송합니다. 이 Node.js 모듈은 SDK for JavaScript에서 `AWS.SES` 클라이언트 클래스의 다음 메서드를 사용하여 이메일 템플릿을 생성하고 사용합니다.

- `createReceiptFilter`
- `listReceiptFilters`
- `deleteReceiptFilter`

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 이러한 작업들을 완료해야 합니다.

- Node.js를 설치합니다. Node.js 설치에 대한 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 공유 자격 증명 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.

SDK 구성

글로벌 구성 객체를 생성한 후 코드에 대한 리전을 설정하여 SDK for JavaScript를 구성합니다. 이 예제에서 리전이 `us-west-2`로 설정되어 있습니다.

```
// Load the SDK for JavaScript
var AWS = require('aws-sdk');
// Set the Region
AWS.config.update({region: 'us-west-2'});
```

IP 주소 필터 생성

이 예제에서는 Node.js 모듈을 사용하여 Amazon SES에서 이메일을 전송합니다. 파일 이름이 `ses_createreceiptfilter.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성합니다.

필터 이름, 필터링할 IP 주소 또는 주소 범위, 필터링된 주소에서 오는 이메일 트래픽을 허용 또는 차단할지 여부를 포함하여 IP 필터를 정의하는 파라미터 값을 전달할 객체를 생성합니다.

`createReceiptFilter` 메서드를 호출하려면 파라미터를 전달하는 Amazon SES 서비스 객체를 호출하기 위한 promise를 생성합니다. 그런 다음 promise 콜백에서 `response`를 처리합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create createReceiptFilter params
var params = {
  Filter: {
    IpFilter: {
      Cidr: "IP_ADDRESS_OR_RANGE",
      Policy: "Allow" | "Block",
    },
    Name: "NAME",
  },
};

// Create the promise and SES service object
var sendPromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .createReceiptFilter(params)
  .promise();

// Handle promise's fulfilled/rejected states
sendPromise
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

예제를 실행하려면 명령줄에서 다음을 입력합니다. Amazon SES에 필터가 생성됩니다.

```
node ses_createreceiptfilter.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

IP 주소 필터 나열

이 예제에서는 Node.js 모듈을 사용하여 Amazon SES에서 이메일을 전송합니다. 파일 이름이 `ses_listreceiptfilters.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성합니다.

비어 있는 파라미터 객체를 생성합니다. `listReceiptFilters` 메서드를 호출하려면 파라미터를 전달하는 Amazon SES 서비스 객체를 호출하기 위한 promise를 생성합니다. 그런 다음 promise 콜백에서 `response`를 처리합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the promise and SES service object
var sendPromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .listReceiptFilters({})
  .promise();

// Handle promise's fulfilled/rejected states
sendPromise
  .then(function (data) {
    console.log(data.Filters);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

예제를 실행하려면 명령줄에서 다음을 입력합니다. Amazon SES가 필터 목록을 반환합니다.

```
node ses_listreceiptfilters.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

IP 주소 필터 삭제하기

이 예제에서는 Node.js 모듈을 사용하여 Amazon SES에서 이메일을 전송합니다. 파일 이름이 `ses_deletereceiptfilter.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성합니다.

삭제할 IP 필터의 이름을 전달할 객체를 생성합니다. `deleteReceiptFilter` 메서드를 호출하려면 파라미터를 전달하는 Amazon SES 서비스 객체를 호출하기 위한 promise를 생성합니다. 그런 다음 promise 콜백에서 `response`를 처리합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the promise and SES service object
var sendPromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .deleteReceiptFilter({ FilterName: "NAME" })
  .promise();

// Handle promise's fulfilled/rejected states
sendPromise
  .then(function (data) {
    console.log("IP Filter deleted");
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

예제를 실행하려면 명령줄에서 다음을 입력합니다. Amazon SES에서 필터가 삭제됩니다.

```
node ses_deletereceiptfilter.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

Amazon SES에서 수신 규칙 사용

이 Node.js 코드 예제는 다음을 보여 줍니다.

- 수신 규칙을 생성하고 삭제합니다.
- 수신 규칙을 수신 규칙 세트로 정리합니다.

Amazon SES의 수신 규칙은 사용자 소유의 이메일 주소 또는 도메인으로 수신된 이메일을 어떻게 처리할지 지정합니다. 수신 규칙에는 조건과 순서 지정된 작업 목록이 포함됩니다. 수신 이메일의 수신자

가 수신 규칙의 조건에 지정된 수신자와 일치하는 경우 Amazon SES는 수신 규칙이 지정하는 작업을 수행합니다.

Amazon SES를 이메일 수신기로 사용하려면 하나 이상의 활성 수신 규칙 세트가 있어야 합니다. 수신 규칙 세트는 확인된 도메인 전체에서 수신하는 메일에 대해 Amazon SES가 수행해야 하는 작업을 지정하는 수신 규칙 모음으로, 순서가 지정되어 있습니다. 자세한 내용은 Amazon Simple Email Service 개발자 안내서의 Amazon SES 이메일 수신 규칙 생성 및 Amazon SES 이메일 수신 규칙 세트 생성 섹션을 참조하세요.

시나리오

이 예제에서는 일련의 Node.js 모듈을 사용하여 다양한 방법으로 이메일을 전송합니다. 이 Node.js 모듈은 SDK for JavaScript에서 `AWS.SES` 클라이언트 클래스의 다음 메서드를 사용하여 이메일 템플릿을 생성하고 사용합니다.

- `createReceiptRule`
- `deleteReceiptRule`
- `createReceiptRuleSet`
- `deleteReceiptRuleSet`

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 이러한 작업들을 완료해야 합니다.

- Node.js를 설치합니다. Node.js 설치에 대한 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 자격 증명 JSON 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.

Amazon S3 수신 규칙 생성

Amazon SES의 각 수신 규칙에는 순서가 지정된 작업 목록이 포함되어 있습니다. 이 예에서는 메일 메시지를 Amazon S3 버킷에 전송하는 Amazon S3 작업이 포함된 수신 규칙을 생성합니다. 수신 규칙 작업에 대한 자세한 내용은 Amazon Simple Email Service 개발자 안내서의 작업 옵션 섹션을 참조하세요.

Amazon SES가 Amazon S3 버킷에 이메일을 쓸 수 있도록 `PutObject` 권한을 Amazon SES에 부여하는 버킷 정책을 생성합니다. 이 버킷 정책 생성에 관한 자세한 내용은 Amazon Simple Email Service 개발자 안내서의 Amazon SES에 S3 버킷에 작성할 수 있는 권한 부여 섹션을 참조하세요.

이 예에서는 Node.js 모듈을 사용해 Amazon SES에 수신 규칙을 생성하여 수신된 메시지를 Amazon S3 버킷에 저장합니다. 파일 이름이 `ses_createreceiptrule.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성합니다.

수신 규칙 세트를 생성하는 데 필요한 값을 전달할 파라미터 객체를 생성합니다. `createReceiptRuleSet` 메서드를 호출하려면 파라미터를 전달하는 Amazon SES 서비스 객체를 호출하기 위한 promise를 생성합니다. 그런 다음 promise 콜백에서 `response`를 처리합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create createReceiptRule params
var params = {
  Rule: {
    Actions: [
      {
        S3Action: {
          BucketName: "S3_BUCKET_NAME",
          ObjectKeyPrefix: "email",
        },
      },
    ],
    Recipients: [
      "DOMAIN | EMAIL_ADDRESS",
      /* more items */
    ],
    Enabled: true | false,
    Name: "RULE_NAME",
    ScanEnabled: true | false,
    TlsPolicy: "Optional",
  },
  RuleSetName: "RULE_SET_NAME",
};

// Create the promise and SES service object
var newRulePromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .createReceiptRule(params)
  .promise();

// Handle promise's fulfilled/rejected states
newRulePromise
```

```
  .then(function (data) {
    console.log("Rule created");
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

예제를 실행하려면 명령줄에서 다음을 입력합니다. Amazon SES가 수신 규칙을 생성합니다.

```
node ses_createreceiptrule.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

수신 규칙 삭제하기

이 예제에서는 Node.js 모듈을 사용하여 Amazon SES에서 이메일을 전송합니다. 파일 이름이 `ses_deletereceiptrule.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성합니다.

삭제할 수신 규칙의 이름을 전달할 파라미터 객체를 생성합니다. `deleteReceiptRule` 메서드를 호출하려면 파라미터를 전달하는 Amazon SES 서비스 객체를 호출하기 위한 promise를 생성합니다. 그런 다음 promise 콜백에서 `response`를 처리합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create deleteReceiptRule params
var params = {
  RuleName: "RULE_NAME" /* required */,
  RuleSetName: "RULE_SET_NAME" /* required */,
};

// Create the promise and SES service object
var newRulePromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .deleteReceiptRule(params)
  .promise();

// Handle promise's fulfilled/rejected states
newRulePromise
  .then(function (data) {
    console.log("Receipt Rule Deleted");
```

```
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

예제를 실행하려면 명령줄에서 다음을 입력합니다. Amazon SES가 수신 규칙 세트 목록을 생성합니다.

```
node ses_deletereceiptrule.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

수신 규칙 세트 생성하기

이 예제에서는 Node.js 모듈을 사용하여 Amazon SES에서 이메일을 전송합니다. 파일 이름이 `ses_createreceiptruleset.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성합니다.

새 수신 규칙 세트의 이름을 전달할 파라미터 객체를 생성합니다. `createReceiptRuleSet` 메서드를 호출하려면 파라미터를 전달하는 Amazon SES 서비스 객체를 호출하기 위한 promise를 생성합니다. 그런 다음 promise 콜백에서 `response`를 처리합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the promise and SES service object
var newRulePromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .createReceiptRuleSet({ RuleSetName: "NAME" })
  .promise();

// Handle promise's fulfilled/rejected states
newRulePromise
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

예제를 실행하려면 명령줄에서 다음을 입력합니다. Amazon SES가 수신 규칙 세트 목록을 생성합니다.

```
node ses_createreceiptruleset.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

수신 규칙 세트 삭제하기

이 예제에서는 Node.js 모듈을 사용하여 Amazon SES에서 이메일을 전송합니다. 파일 이름이 `ses_deletereceiptruleset.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성합니다.

삭제할 수신 규칙 세트의 이름을 전달할 객체를 생성합니다. `deleeReceiptRuleSet` 메서드를 호출하려면 파라미터를 전달하는 Amazon SES 서비스 객체를 호출하기 위한 promise를 생성합니다. 그런 다음 promise 콜백에서 `response`를 처리합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the promise and SES service object
var newRulePromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .deleteReceiptRuleSet({ RuleSetName: "NAME" })
  .promise();

// Handle promise's fulfilled/rejected states
newRulePromise
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

예제를 실행하려면 명령줄에서 다음을 입력합니다. Amazon SES가 수신 규칙 세트 목록을 생성합니다.

```
node ses_deletereceiptruleset.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

Amazon Simple Notification Service 예

Amazon Simple Notification Service(Amazon SNS)는 구독 중인 엔드포인트 또는 클라이언트에 대한 메시지 전달 또는 전송을 조정 및 관리하는 웹 서비스입니다.

Amazon SNS에는 게시자와 구독자 또는 생산자와 소비자라고 하는 두 가지 클라이언트 유형이 있습니다.

게시자는 주제에 대한 메시지를 생산 및 발송함으로써 구독자와 비동시적으로 통신하는 논리적 액세스 및 커뮤니케이션 채널입니다. 구독자(웹 서버, 이메일 주소, Amazon SQS 대기열, Lambda 함수)는 주제를 구독할 때 지원되는 프로토콜(Amazon SQS, HTTP/S, 이메일, SMS, AWS Lambda) 중 하나를 통해 메시지 또는 알림을 소비하거나 수신합니다.

Amazon SNS용 JavaScript API는 `Class: AWS.SNS`를 통해 노출됩니다.

주제

- Amazon SNS에서 주제 관리
- Amazon SNS에서 메시지 게시
- Amazon SNS에서 구독 관리
- Amazon SNS를 통한 SMS 메시지 전송

Amazon SNS에서 주제 관리

이 Node.js 코드 예제는 다음을 보여 줍니다.

- Amazon SNS에서 알림을 게시할 수 있는 주제를 생성하는 방법
- Amazon SNS에서 생성된 주제를 삭제하는 방법
- 사용 가능한 주제 목록을 가져오는 방법.
- 주제 속성을 가져오고 설정하는 방법.

시나리오

이 예에서는 일련의 Node.js 모듈을 사용하여 Amazon SNS 주제를 생성, 나열 및 삭제하고 주제 속성을 처리합니다. Node.js 모듈은 SDK for JavaScript로 `AWS.SNS` 클라이언트 클래스의 다음 메서드를 사용하여 주제를 관리합니다.

- `createTopic`
- `listTopics`
- `deleteTopic`
- `getTopicAttributes`
- `setTopicAttributes`

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 이러한 작업들을 완료해야 합니다.

- Node.js를 설치합니다. Node.js 설치에 대한 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 자격 증명 JSON 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.

주제 생성

이 예에서는 Node.js 모듈을 사용하여 Amazon SNS 주제를 생성합니다. 파일 이름이 `sns_createtopic.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성합니다.

`Name` 클라이언트 클래스의 `createTopic` 메서드에 새 주제의 `AWS.SNS`을 전달할 객체를 생성합니다. `createTopic` 메서드를 직접 호출하려면 파라미터 객체를 전달하는 Amazon SNS 서비스 객체를 간접 호출하기 위한 promise를 생성합니다. 그런 다음 promise 콜백에서 `response`를 처리합니다. promise에서 반환되는 `data`에는 주제의 ARN이 포함됩니다.

```
// Load the AWS SDK for Node.js
```

```
var AWS = require("aws-sdk");
// Set region
AWS.config.update({ region: "REGION" });

// Create promise and SNS service object
var createTopicPromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .createTopic({ Name: "TOPIC_NAME" })
  .promise();

// Handle promise's fulfilled/rejected states
createTopicPromise
  .then(function (data) {
    console.log("Topic ARN is " + data.TopicArn);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node sns_createtopic.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

주제 나열

이 예에서는 Node.js 모듈을 사용하여 모든 Amazon SNS 주제를 나열합니다. 파일 이름이 `sns_listtopics.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성합니다.

`listTopics` 클라이언트 클래스의 `AWS.SNS` 메서드에 전달할 비어 있는 객체를 생성합니다. `listTopics` 메서드를 직접 호출하려면 파라미터 객체를 전달하는 Amazon SNS 서비스 객체를 간접 호출하기 위한 promise를 생성합니다. 그런 다음 promise 콜백에서 `response`를 처리합니다. promise에서 반환되는 `data`에는 주제 ARN의 배열이 포함됩니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set region
AWS.config.update({ region: "REGION" });

// Create promise and SNS service object
var listTopicsPromise = new AWS.SNS({ apiVersion: "2010-03-31" })
```

```
  .listTopics({})
  .promise();

// Handle promise's fulfilled/rejected states
listTopicsPromise
  .then(function (data) {
    console.log(data.Topics);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node sns_listtopics.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

주제 삭제

이 예에서는 Node.js 모듈을 사용하여 Amazon SNS 주제를 삭제합니다. 파일 이름이 `sns_deletetopic.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성합니다.

`TopicArn` 클라이언트 클래스의 `deleteTopic` 메서드에 전달할 삭제할 주제의 `AWS.SNS`을 포함하는 객체를 생성합니다. `deleteTopic` 메서드를 직접 호출하려면 파라미터 객체를 전달하는 Amazon SNS 서비스 객체를 간접 호출하기 위한 promise를 생성합니다. 그런 다음 promise 콜백에서 `response`를 처리합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set region
AWS.config.update({ region: "REGION" });

// Create promise and SNS service object
var deleteTopicPromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .deleteTopic({ TopicArn: "TOPIC_ARN" })
  .promise();

// Handle promise's fulfilled/rejected states
deleteTopicPromise
  .then(function (data) {
    console.log("Topic Deleted");
```

```
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node sns_deletetopic.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

주제 속성 가져오기

이 예에서는 Node.js 모듈을 사용하여 Amazon SNS 주제의 속성을 검색합니다. 파일 이름이 `sns_gettopicattributes.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성합니다.

`TopicArn` 클라이언트 클래스의 `getTopicAttributes` 메서드에 전달할 삭제할 주제의 `AWS.SNS`을 포함하는 객체를 생성합니다. `getTopicAttributes` 메서드를 직접 호출하려면 파라미터 객체를 전달하는 Amazon SNS 서비스 객체를 간접 호출하기 위한 promise를 생성합니다. 그런 다음 promise 콜백에서 `response`를 처리합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set region
AWS.config.update({ region: "REGION" });

// Create promise and SNS service object
var getTopicAttribsPromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .getTopicAttributes({ TopicArn: "TOPIC_ARN" })
  .promise();

// Handle promise's fulfilled/rejected states
getTopicAttribsPromise
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node sns_gettopicattributes.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

주제 속성 설정

이 예에서는 Node.js 모듈을 사용하여 Amazon SNS 주제의 변경 가능한 속성을 설정합니다. 파일 이름이 `sns_settopicattributes.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성합니다.

속성을 설정하려고 하는 주제의 `TopicArn`, 설정할 속성의 이름, 해당 속성의 새 값을 포함하여 속성 업데이트를 위한 파라미터를 포함하는 객체를 생성합니다. `Policy`, `DisplayName` 및 `DeliveryPolicy` 속성만 설정할 수 있습니다. `setTopicAttributes` 클라이언트 클래스의 `AWS.SNS` 메서드에 파라미터를 전달합니다. `setTopicAttributes` 메서드를 직접 호출하려면 파라미터 객체를 전달하는 Amazon SNS 서비스 객체를 간접 호출하기 위한 promise를 생성합니다. 그런 다음 promise 콜백에서 `response`를 처리합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set region
AWS.config.update({ region: "REGION" });

// Create setTopicAttributes parameters
var params = {
  AttributeName: "ATTRIBUTE_NAME" /* required */,
  TopicArn: "TOPIC_ARN" /* required */,
  AttributeValue: "NEW_ATTRIBUTE_VALUE",
};

// Create promise and SNS service object
var setTopicAttribsPromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .setTopicAttributes(params)
  .promise();

// Handle promise's fulfilled/rejected states
setTopicAttribsPromise
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node sns_settopicattributes.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

Amazon SNS에서 메시지 게시

이 Node.js 코드 예제는 다음을 보여 줍니다.

- Amazon SNS 주제에 메시지를 게시하는 방법

시나리오

이 예에서는 일련의 Node.js 모듈을 사용하여 Amazon SNS의 메시지를 주제 엔드포인트, 이메일 또는 전화번호에 게시합니다. Node.js 모듈은 SDK for JavaScript로 `AWS.SNS` 클라이언트 클래스의 다음 메서드를 사용하여 메시지를 전송합니다.

- `publish`

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 이러한 작업들을 완료해야 합니다.

- Node.js를 설치합니다. Node.js 설치에 대한 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 자격 증명 JSON 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.

Amazon SNS 주제에 메시지 게시

이 예에서는 Node.js 모듈을 사용하여 Amazon SNS 주제에 메시지를 게시합니다. 파일 이름이 `sns_publishtotopic.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성합니다.

메시지 텍스트 및 Amazon SNS 주제의 ARN을 포함하여 메시지를 게시하기 위한 파라미터를 포함하는 객체를 생성합니다. 사용 가능한 SMS 속성에 대한 세부 정보는 SetSMSAttributes를 참조하세요.

`publish` 클라이언트 클래스의 `AWS.SNS` 메서드에 파라미터를 전달합니다. 파라미터 객체를 전달하는 Amazon SNS 서비스 객체를 호출하기 위한 promise를 생성합니다. 그런 다음 promise 콜백에서 response를 처리합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set region
AWS.config.update({ region: "REGION" });

// Create publish parameters
var params = {
  Message: "MESSAGE_TEXT" /* required */,
  TopicArn: "TOPIC_ARN",
};

// Create promise and SNS service object
var publishTextPromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .publish(params)
  .promise();

// Handle promise's fulfilled/rejected states
publishTextPromise
  .then(function (data) {
    console.log(
      `Message ${params.Message} sent to the topic ${params.TopicArn}`
    );
    console.log("MessageID is " + data.MessageId);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node sns_publishtotopic.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

Amazon SNS에서 구독 관리

이 Node.js 코드 예제는 다음을 보여 줍니다.

- Amazon SNS 주제의 모든 구독을 나열하는 방법
- 이메일 주소, 애플리케이션 엔드포인트 또는 AWS Lambda 함수에서 Amazon SNS 주제를 구독하는 방법
- Amazon SNS 주제의 구독을 취소하는 방법

시나리오

이 예에서는 일련의 Node.js 모듈을 사용하여 Amazon SNS 주제에 알림 메시지를 게시합니다. Node.js 모듈은 SDK for JavaScript로 `AWS.SNS` 클라이언트 클래스의 다음 메서드를 사용하여 주제를 관리합니다.

- `subscribe`
- `confirmSubscription`
- `listSubscriptionsByTopic`
- `unsubscribe`

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 이러한 작업들을 완료해야 합니다.

- Node.js를 설치합니다. Node.js 설치에 대한 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 자격 증명 JSON 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.

주제에 대한 구독 나열

이 예에서는 Node.js 모듈을 사용하여 Amazon SNS 주제에 대한 모든 구독을 나열합니다. 파일 이름이 `sns_listsubscriptions.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성합니다.

구독을 나열할 주제에 대한 TopicArn 파라미터를 포함하는 객체를 생성합니다. listSubscriptionsByTopic 클라이언트 클래스의 AWS.SNS 메서드에 파라미터를 전달합니다. listSubscriptionsByTopic 메서드를 직접 호출하려면 파라미터 객체를 전달하는 Amazon SNS 서비스 객체를 간접 호출하기 위한 promise를 생성합니다. 그런 다음 promise 콜백에서 response를 처리합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set region
AWS.config.update({ region: "REGION" });

const params = {
  TopicArn: "TOPIC_ARN",
};

// Create promise and SNS service object
var subslistPromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .listSubscriptionsByTopic(params)
  .promise();

// Handle promise's fulfilled/rejected states
subslistPromise
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node sns_listsubscriptions.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

이메일 주소에서 주제 구독

이 예에서는 Node.js 모듈을 사용하여 이메일 주소에서 Amazon SNS 주제의 SMTP 이메일 메시지를 수신하도록 이메일 주소에서 주제를 구독합니다. 파일 이름이 sns_subscribeemail.js인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성합니다.

`Protocol` 프로토콜, 구독할 주제의 `email`, 메시지 `TopicArn`로 사용되는 이메일 주소를 지정하기 위한 `Endpoint` 파라미터를 포함하는 객체를 생성합니다. `subscribe` 클라이언트 클래스의 `AWS.SNS` 메서드에 파라미터를 전달합니다. 이 항목의 다른 예에 나와 있듯이, `subscribe` 메서드를 사용하면 전달된 파라미터에 사용되는 값에 따라 여러 다양한 엔드포인트에서 Amazon SNS 주제를 구독할 수 있습니다.

`subscribe` 메서드를 직접 호출하려면 파라미터 객체를 전달하는 Amazon SNS 서비스 객체를 간접 호출하기 위한 promise를 생성합니다. 그런 다음 promise 콜백에서 `response`를 처리합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set region
AWS.config.update({ region: "REGION" });

// Create subscribe/email parameters
var params = {
  Protocol: "EMAIL" /* required */,
  TopicArn: "TOPIC_ARN" /* required */,
  Endpoint: "EMAIL_ADDRESS",
};

// Create promise and SNS service object
var subscribePromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .subscribe(params)
  .promise();

// Handle promise's fulfilled/rejected states
subscribePromise
  .then(function (data) {
    console.log("Subscription ARN is " + data.SubscriptionArn);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node sns_subscribeemail.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

애플리케이션 엔드포인트에서 주제 구독

이 예에서는 Node.js 모듈을 사용하여 모바일 애플리케이션 엔드포인트에서 Amazon SNS 주제의 알림을 수신하도록 모바일 애플리케이션 엔드포인트에서 주제를 구독합니다. 파일 이름이 `sns_subscribeapp.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성합니다.

`Protocol` 프로토콜, 구독할 주제의 `application`, `TopicArn` 파라미터에 대한 모바일 애플리케이션 엔드포인트의 ARN을 지정하기 위한 `Endpoint` 파라미터를 포함하는 객체를 생성합니다. `subscribe` 클라이언트 클래스의 `AWS.SNS` 메서드에 파라미터를 전달합니다.

`subscribe` 메서드를 직접 호출하려면 파라미터 객체를 전달하는 Amazon SNS 서비스 객체를 간접 호출하기 위한 promise를 생성합니다. 그런 다음 promise 콜백에서 `response`를 처리합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set region
AWS.config.update({ region: "REGION" });

// Create subscribe/email parameters
var params = {
  Protocol: "application" /* required */,
  TopicArn: "TOPIC_ARN" /* required */,
  Endpoint: "MOBILE_ENDPOINT_ARN",
};

// Create promise and SNS service object
var subscribePromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .subscribe(params)
  .promise();

// Handle promise's fulfilled/rejected states
subscribePromise
  .then(function (data) {
    console.log("Subscription ARN is " + data.SubscriptionArn);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node sns_subscribeapp.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

Lambda 함수에서 주제 구독

이 예에서는 Node.js 모듈을 사용하여 AWS Lambda 함수에서 Amazon SNS 주제의 알림을 수신하도록 Lambda 함수에서 주제를 구독합니다. 파일 이름이 `sns_subscribelambda.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성합니다.

`Protocol` 프로토콜, 구독할 주제의 `lambda`, `TopicArn` 파라미터로 사용되는 AWS Lambda 함수의 ARN을 지정하는 `Endpoint` 파라미터를 포함하는 객체를 생성합니다. `subscribe` 클라이언트 클래스의 `AWS.SNS` 메서드에 파라미터를 전달합니다.

`subscribe` 메서드를 직접 호출하려면 파라미터 객체를 전달하는 Amazon SNS 서비스 객체를 간접 호출하기 위한 promise를 생성합니다. 그런 다음 promise 콜백에서 `response`를 처리합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set region
AWS.config.update({ region: "REGION" });

// Create subscribe/email parameters
var params = {
  Protocol: "lambda" /* required */,
  TopicArn: "TOPIC_ARN" /* required */,
  Endpoint: "LAMBDA_FUNCTION_ARN",
};

// Create promise and SNS service object
var subscribePromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .subscribe(params)
  .promise();

// Handle promise's fulfilled/rejected states
subscribePromise
  .then(function (data) {
    console.log("Subscription ARN is " + data.SubscriptionArn);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node sns_subscribelambda.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

주제의 구독 취소

이 예에서는 Node.js 모듈을 사용하여 Amazon SNS 주제 구독을 취소합니다. 파일 이름이 `sns_unsubscribe.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성합니다.

구독 해제할 구독의 ARN을 지정하는 `SubscriptionArn` 파라미터를 포함하는 객체를 생성합니다. `unsubscribe` 클라이언트 클래스의 `AWS.SNS` 메서드에 파라미터를 전달합니다.

`unsubscribe` 메서드를 직접 호출하려면 파라미터 객체를 전달하는 Amazon SNS 서비스 객체를 간접 호출하기 위한 promise를 생성합니다. 그런 다음 promise 콜백에서 `response`를 처리합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set region
AWS.config.update({ region: "REGION" });

// Create promise and SNS service object
var subscribePromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .unsubscribe({ SubscriptionArn: TOPIC_SUBSCRIPTION_ARN })
  .promise();

// Handle promise's fulfilled/rejected states
subscribePromise
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node sns_unsubscribe.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

Amazon SNS를 통한 SMS 메시지 전송

이 Node.js 코드 예제는 다음을 보여 줍니다.

- Amazon SNS에 대한 SMS 메시징 기본 설정을 가져오고 설정하는 방법
- 전화번호를 점검하여 SMS 메시지 수신을 옵트아웃했는지 여부를 확인하는 방법.
- SMS 메시지 수신을 옵트아웃한 전화번호의 목록을 가져오는 방법.
- SMS 메시지를 전송하는 방법.

시나리오

사용자는 Amazon SNS를 사용하여 SMS 수신 가능한 디바이스에 문자 메시지 또는 SMS 메시지를 전송할 수 있습니다. 전화번호로 메시지를 직접 전송할 수 있으며, 전화번호에서 주제를 구독하고 메시지를 주제로 전송하여 메시지를 여러 전화번호로 한 번에 전송할 수 있습니다.

이 예에서는 일련의 Node.js 모듈을 사용하여 Amazon SNS의 SMS 문자 메시지를 SMS 지원 디바이스에 게시합니다. Node.js 모듈은 SDK for JavaScript로 `AWS.SNS` 클라이언트 클래스의 다음 메서드를 사용하여 SMS 메시지를 게시합니다.

- `getSMSAttributes`
- `setSMSAttributes`
- `checkIfPhoneNumberIsOptedOut`
- `listPhoneNumbersOptedOut`
- `publish`

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 이러한 작업들을 완료해야 합니다.

- Node.js를 설치합니다. Node.js 설치에 대한 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 자격 증명 JSON 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.

SMS 속성 가져오기

Amazon SNS를 사용하여 전송을 최적화하는 방법(비용 또는 안정성 있는 전송), 월 지출 한도, 메시지 전송을 로깅하는 방법, 일일 SMS 사용 보고서를 구독하는지 여부 등 SMS 메시징에 대한 기본 설정을 지정합니다. 이러한 기본 설정을 검색하여 Amazon SNS의 SMS 속성으로 설정합니다.

이 예에서는 Node.js 모듈을 사용하여 Amazon SNS에서 현재 SMS 속성을 가져옵니다. 파일 이름이 `sns_getsmstype.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성합니다. 가져올 개별 속성의 이름을 포함하여 SMS 속성을 가져오기 위한 파라미터를 포함하는 객체를 생성합니다. 사용 가능한 SMS 속성에 대한 자세한 내용은 Amazon Simple Notification Service API 참조의 SetSMSAttributes를 참조하세요.

이 예제에서는 SMS 메시지를 최저 비용이 발생하도록 메시지 전송을 최적화하는 `DefaultSMSType`로 전송할지 또는 최고 안정성을 달성하도록 메시지 전송을 최적화하는 `Promotional`로 전송할지를 제어하는 `Transactional` 속성을 가져옵니다. `setTopicAttributes` 클라이언트 클래스의 `AWS.SNS` 메서드에 파라미터를 전달합니다. `getSMSAttributes` 메서드를 직접 호출하려면 파라미터 객체를 전달하는 Amazon SNS 서비스 객체를 간접 호출하기 위한 promise를 생성합니다. 그런 다음 promise 콜백에서 `response`를 처리합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set region
AWS.config.update({ region: "REGION" });

// Create SMS Attribute parameter you want to get
var params = {
  attributes: [
    "DefaultSMSType",
    "ATTRIBUTE_NAME",
    /* more items */
  ],
};

// Create promise and SNS service object
var getSMSTypePromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .getSMSAttributes(params)
  .promise();

// Handle promise's fulfilled/rejected states
getSMSTypePromise
```

```
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node sns_getsmstype.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

SMS 속성 설정

이 예에서는 Node.js 모듈을 사용하여 Amazon SNS에서 현재 SMS 속성을 가져옵니다. 파일 이름이 `sns_setsmstype.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성합니다. 설정할 개별 속성의 이름과 각 속성에 설정할 값을 포함하여 SMS 속성을 설정하기 위한 파라미터를 포함하는 객체를 생성합니다. 사용 가능한 SMS 속성에 대한 자세한 내용은 Amazon Simple Notification Service API 참조의 SetSMSAttributes를 참조하세요.

다음 예제에서는 `DefaultSMSType` 속성을 `Transactional`로 설정하여 최고의 안정성을 달성하도록 메시지 전송을 최적화합니다. `setTopicAttributes` 클라이언트 클래스의 `AWS.SNS` 메서드에 파라미터를 전달합니다. `getSMSAttributes` 메서드를 직접 호출하려면 파라미터 객체를 전달하는 Amazon SNS 서비스 객체를 간접 호출하기 위한 promise를 생성합니다. 그런 다음 promise 콜백에서 `response`를 처리합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set region
AWS.config.update({ region: "REGION" });

// Create SMS Attribute parameters
var params = {
  attributes: {
    /* required */
    DefaultSMSType: "Transactional" /* highest reliability */,
    //'DefaultSMSType': 'Promotional' /* lowest cost */
  },
};
```

```
// Create promise and SNS service object
var setSMSTypePromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .setSMSAttributes(params)
  .promise();

// Handle promise's fulfilled/rejected states
setSMSTypePromise
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node sns_setsmstype.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

전화번호가 옵트아웃되었는지 여부 확인

이 예제에서는 Node.js 모듈을 사용하여 전화 번호가 SMS 메시지 수신에서 옵트아웃되었는지 여부를 확인합니다. 파일 이름이 `sns_checkphoneoptout.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성합니다. 파라미터로 확인할 전화번호를 포함하는 객체를 생성합니다.

이 예제에서는 확인할 전화번호를 지정하는 `PhoneNumber` 파라미터를 설정합니다. `checkIfPhoneNumberIsOptedOut` 클라이언트 클래스의 `AWS.SNS` 메서드에 객체를 전달합니다. `checkIfPhoneNumberIsOptedOut` 메서드를 직접 호출하려면 파라미터 객체를 전달하는 Amazon SNS 서비스 객체를 간접 호출하기 위한 promise를 생성합니다. 그런 다음 promise 콜백에서 `response`를 처리합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set region
AWS.config.update({ region: "REGION" });

// Create promise and SNS service object
var phonenumPromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .checkIfPhoneNumberIsOptedOut({ phoneNumber: "PHONE_NUMBER" })
  .promise();
```

```
// Handle promise's fulfilled/rejected states
phonenumPromise
  .then(function (data) {
    console.log("Phone Opt Out is " + data.isOptedOut);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node sns_checkphoneoptout.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

옵트아웃된 전화번호 나열

이 예제에서는 Node.js 모듈을 사용하여 SMS 메시지 수신에서 옵트아웃한 전화번호의 목록을 가져옵니다. 파일 이름이 `sns_listnumbersoptedout.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성합니다. 비어 있는 객체를 파라미터로 생성합니다.

`listPhoneNumbersOptedOut` 클라이언트 클래스의 `AWS.SNS` 메서드에 객체를 전달합니다. `listPhoneNumbersOptedOut` 메서드를 직접 호출하려면 파라미터 객체를 전달하는 Amazon SNS 서비스 객체를 간접 호출하기 위한 promise를 생성합니다. 그런 다음 promise 콜백에서 `response`를 처리합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set region
AWS.config.update({ region: "REGION" });

// Create promise and SNS service object
var phonelistPromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .listPhoneNumbersOptedOut({})
  .promise();

// Handle promise's fulfilled/rejected states
phonelistPromise
  .then(function (data) {
    console.log(data);
```

```
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node sns_listnumbersoptedout.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

SMS 메시지 게시

이 예제에서는 Node.js 모듈을 사용하여 SMS 메시지를 전화번호에 전송합니다. 파일 이름이 `sns_publishsms.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성합니다. `Message` 및 `PhoneNumber` 파라미터를 포함하는 객체를 생성합니다.

SMS 메시지를 전송할 때 E.164 형식을 사용하여 전화번호를 지정합니다. E.164는 국제 통신에 사용되는 전화번호 구조의 표준입니다. 이 형식을 따르는 전화번호는 최대 15자리 숫자를 사용할 수 있으며 더하기 문자(+) 및 국가 코드가 접두사로 추가됩니다. 예를 들어, E.164 형식의 미국 전화번호는 +1001XXX5550100으로 표시될 수 있습니다.

이 예제에서는 메시지를 전송할 전화번호를 지정하는 `PhoneNumber` 파라미터를 설정합니다. `publish` 클라이언트 클래스의 `AWS.SNS` 메서드에 객체를 전달합니다. `publish` 메서드를 직접 호출하려면 파라미터 객체를 전달하는 Amazon SNS 서비스 객체를 간접 호출하기 위한 promise를 생성합니다. 그런 다음 promise 콜백에서 `response`를 처리합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set region
AWS.config.update({ region: "REGION" });

// Create publish parameters
var params = {
  Message: "TEXT_MESSAGE" /* required */,
  PhoneNumber: "E.164_PHONE_NUMBER",
};

// Create promise and SNS service object
var publishTextPromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .publish(params)
```

```
  .promise();

// Handle promise's fulfilled/rejected states
publishTextPromise
  .then(function (data) {
    console.log("MessageID is " + data.MessageId);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node sns_publishsms.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

Amazon SQS 예제

Amazon Simple Queue Service(Amazon SQS)는 빠르고 안정적이며 확장 가능한 완전관리형 메시지 대기열 서비스입니다. Amazon SQS를 사용하면 클라우드 애플리케이션의 구성 요소를 분리할 수 있습니다. Amazon SQS에는 높은 처리량과 최소 1회 처리 기능을 갖춘 표준 대기열과 선입선출(FIFO) 전송 및 정확히 1회 처리를 제공하는 FIFO 대기열이 포함되어 있습니다.

Amazon SQS용 JavaScript API는 `AWS.SQS` 클라이언트 클래스를 통해 노출됩니다. Amazon SQS 클라이언트 클래스 사용에 대한 자세한 내용은 API 참조의 `Class: AWS.SQS` 섹션을 참조하세요.

주제

- Amazon SQS에서 대기열 사용

- Amazon SQS에서 메시지 전송 및 수신
- Amazon SQS의 제한 시간 초과 관리
- Amazon SQS에서 긴 폴링 활성화
- Amazon SQS에서 DLQ(Dead Letter Queue) 사용

Amazon SQS에서 대기열 사용

이 Node.js 코드 예제는 다음을 보여 줍니다.

- 모든 메시지 대기열의 목록을 가져오는 방법
- 특정 대기열의 URL을 획득하는 방법
- 대기열을 생성하고 삭제하는 방법

예제 소개

이 예제에서는 일련의 Node.js 모듈을 사용하여 대기열을 작업합니다. Node.js 모듈은 SDK for JavaScript로 `AWS.SQS` 클라이언트 클래스의 다음 메서드를 직접 호출하기 위해 대기열을 활성화합니다.

- `listQueues`
- `createQueue`
- `getQueueUrl`
- `deleteQueue`

Amazon SQS 메시지에 대한 자세한 정보는 Amazon Simple Queue Service 개발자 안내서의 대기열의 작동 방식을 참조하세요.

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 이러한 작업들을 완료해야 합니다.

- Node.js를 설치합니다. Node.js 설치에 대한 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 공유 자격 증명 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.

대기열 목록 표시

파일 이름이 `sqs_listqueues.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. Amazon SQS에 액세스하려면 `AWS.SQS` 서비스 객체를 생성합니다. 대기열 목록을 표시하는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. 이 객체는 기본적으로 비어 있는 객체입니다. `listQueues` 메서드를 호출하여 대기열 목록을 검색합니다. 콜백에서 모든 대기열의 URL이 반환됩니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create an SQS service object
var sqs = new AWS.SQS({ apiVersion: "2012-11-05" });

var params = {};

sqs.listQueues(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.QueueUrls);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node sqs_listqueues.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

대기열 만들기

파일 이름이 `sqs_createqueue.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. Amazon SQS에 액세스하려면 `AWS.SQS` 서비스 객체를 생성합니다. 대기열 목록을 표시하는 데 필

요한 파라미터를 포함하는 JSON 객체를 생성합니다. 이 객체에는 생성된 대기열의 이름이 포함되어야 합니다. 파라미터에는 메시지 전송이 지연되는 시간(초) 또는 수신한 메시지를 보관할 시간(초)과 같은 대기열의 속성도 포함될 수 있습니다. `createQueue` 메서드를 호출합니다. 콜백에서 생성된 대기열의 URL이 반환됩니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create an SQS service object
var sqs = new AWS.SQS({ apiVersion: "2012-11-05" });

var params = {
  QueueName: "SQS_QUEUE_NAME",
  Attributes: {
    DelaySeconds: "60",
    MessageRetentionPeriod: "86400",
  },
};

sqs.createQueue(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.QueueUrl);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node sqs_createqueue.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

대기열의 URL 가져오기

파일 이름이 `sqs_getqueueurl.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. Amazon SQS에 액세스하려면 `AWS.SQS` 서비스 객체를 생성합니다. 대기열 목록을 표시하는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. 이 객체에는 URL을 가져올 대기열의 이름이 포함되어야 합니다. `getQueueUrl` 메서드를 호출합니다. 콜백에서 지정된 대기열의 URL이 반환됩니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create an SQS service object
var sqs = new AWS.SQS({ apiVersion: "2012-11-05" });

var params = {
  QueueName: "SQS_QUEUE_NAME",
};

sqs.getQueueUrl(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.QueueUrl);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node sqs_getqueueurl.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

대기열 삭제

파일 이름이 `sqs_deletequeue.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. Amazon SQS에 액세스하려면 `AWS.SQS` 서비스 객체를 생성합니다. 대기열을 삭제하는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. 이 객체는 삭제하려는 대기열의 URL로 구성됩니다. `deleteQueue` 메서드를 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create an SQS service object
var sqs = new AWS.SQS({ apiVersion: "2012-11-05" });
```

```
var params = {
  QueueUrl: "SQS_QUEUE_URL",
};

sqs.deleteQueue(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node sqs_deletequeue.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

Amazon SQS에서 메시지 전송 및 수신

이 Node.js 코드 예제는 다음을 보여 줍니다.

- 대기열에 있는 메시지를 전송하는 방법.
- 대기열에 있는 메시지를 수신하는 방법.
- 대기열에 있는 메시지를 삭제하는 방법.

시나리오

이 예제에서는 일련의 Node.js 모듈을 사용하여 메시지를 전송하고 수신합니다. Node.js 모듈은 SDK for JavaScript로 `AWS.SQS` 클라이언트 클래스의 다음 메서드를 사용하여 메시지를 전송하고 수신합니다.

- `sendMessage`
- `receiveMessage`

- deleteMessage

Amazon SQS 메시지에 대한 자세한 내용은 Amazon Simple Queue Service 개발자 안내서의 Amazon SQS 대기열로 메시지 전송 및 Amazon SQS 대기열에서 메시지 수신 및 삭제를 참조하세요.

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 이러한 작업들을 완료해야 합니다.

- Node.js를 설치합니다. Node.js 설치에 대한 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 공유 자격 증명 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.
- Amazon SQS 대기열을 생성합니다. 대기열 생성에 대한 예제는 Amazon SQS에서 대기열 사용 섹션을 참조하세요.

대기열로 메시지 전송

파일 이름이 `sqs_sendmessage.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. Amazon SQS에 액세스하려면 `AWS.SQS` 서비스 객체를 생성합니다. 이 메시지를 전송하려는 대기열의 URL을 포함하여 메시지에 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. 이 예제에서 메시지는 제목, 저자, 목록에 속해 있는 기간(주)을 포함하여 소설 베스트셀러 목록에 있는 책에 대한 세부 정보를 제공합니다.

`sendMessage` 메서드를 호출합니다. 콜백에서 메시지의 고유 ID가 반환됩니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create an SQS service object
var sqs = new AWS.SQS({ apiVersion: "2012-11-05" });

var params = {
  // Remove DelaySeconds parameter and value for FIFO queues
  DelaySeconds: 10,
  MessageAttributes: {
    Title: {
      DataType: "String",
      StringValue: "The Whistler",
```

```
    },
    Author: {
      DataType: "String",
      StringValue: "John Grisham",
    },
    WeeksOn: {
      DataType: "Number",
      StringValue: "6",
    },
  },
  MessageBody:
    "Information about current NY Times fiction bestseller for week of 12/11/2016.",
  // MessageDeduplicationId: "TheWhistler",  // Required for FIFO queues
  // MessageGroupId: "Group1",  // Required for FIFO queues
  QueueUrl: "SQS_QUEUE_URL",
};

sqs.sendMessage(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.MessageId);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node sqs_sendmessage.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

대기열에서 메시지 수신 및 삭제

파일 이름이 `sqs_receivemessage.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. Amazon SQS에 액세스하려면 `AWS.SQS` 서비스 객체를 생성합니다. 메시지를 수신하려는 대기열의 URL을 포함하여 메시지에 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. 이 예제에서 파라미터는 모든 메시지 속성의 수신 및 10개 이하의 메시지 수신을 지정합니다.

`receiveMessage` 메서드를 호출합니다. 콜백에서 나중에 해당 메시지를 삭제하기 위해 사용하는 각 메시지의 `ReceiptHandle`을 검색할 수 있는 `Message` 객체의 배열이 반환됩니다. 메시지를 삭제하는 데 필요한 파라미터를 포함하는 또 다른 JSON 객체를 생성합니다. 이 파라미터는 대기열의 URL 및 `ReceiptHandle` 값입니다. `deleteMessage` 메서드를 호출하여 수신한 메시지를 삭제합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create an SQS service object
var sqs = new AWS.SQS({ apiVersion: "2012-11-05" });

var queueURL = "SQS_QUEUE_URL";

var params = {
  AttributeNames: ["SentTimestamp"],
  MaxNumberOfMessages: 10,
  MessageAttributeNames: ["All"],
  QueueUrl: queueURL,
  VisibilityTimeout: 20,
  WaitTimeSeconds: 0,
};

sqs.receiveMessage(params, function (err, data) {
  if (err) {
    console.log("Receive Error", err);
  } else if (data.Messages) {
    var deleteParams = {
      QueueUrl: queueURL,
      ReceiptHandle: data.Messages[0].ReceiptHandle,
    };
    sqs.deleteMessage(deleteParams, function (err, data) {
      if (err) {
        console.log("Delete Error", err);
      } else {
        console.log("Message Deleted", data);
      }
    });
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node sqs_receivemessage.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

Amazon SQS의 제한 시간 초과 관리

이 Node.js 코드 예제는 다음을 보여 줍니다.

- 대기열에서 수신된 메시지를 볼 수 없는 시간 간격을 지정하는 방법

시나리오

이 예제에서는 Node.js 모듈을 사용하여 제한 시간 초과를 관리합니다. Node.js 모듈은 SDK for JavaScript로 `AWS.SQS` 클라이언트 클래스의 다음 메서드를 사용하여 제한 시간 초과를 관리합니다.

- `changeMessageVisibility`

Amazon SQS 가시성 제한 시간에 대한 자세한 정보는 Amazon Simple Queue Service 개발자 안내서의 제한 시간 초과를 참조하세요.

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 이러한 작업들을 완료해야 합니다.

- Node.js를 설치합니다. Node.js 설치에 대한 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 공유 자격 증명 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.
- Amazon SQS 대기열을 생성합니다. 대기열 생성에 대한 예제는 Amazon SQS에서 대기열 사용 섹션을 참조하세요.
- 대기열로 메시지를 전송합니다. 대기열로 메시지를 전송하는 방법에 대한 예제는 Amazon SQS에서 메시지 전송 및 수신 섹션을 참조하세요.

제한 시간 초과 변경

파일 이름이 `sqs_changingvisibility.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. Amazon Simple Queue Service에 액세스하려면 `AWS.SQS` 서비스 객체를 생성합니다. 대기열에서 메시지를 검색합니다.

대기열에서 메시지를 수신하면 메시지가 포함된 대기열의 URL, 메시지를 수신할 때 반환된 `ReceiptHandle`, 새 제한 시간(초)을 포함하여 제한 시간을 설정하기 위해 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. `changeMessageVisibility` 메서드를 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region to us-west-2
AWS.config.update({ region: "us-west-2" });

// Create the SQS service object
var sqs = new AWS.SQS({ apiVersion: "2012-11-05" });

var queueURL = "https://sqs.REGION.amazonaws.com/ACCOUNT-ID/QUEUE-NAME";

var params = {
  AttributeNames: ["SentTimestamp"],
  MaxNumberOfMessages: 1,
  MessageAttributeNames: ["All"],
  QueueUrl: queueURL,
};

sqs.receiveMessage(params, function (err, data) {
  if (err) {
    console.log("Receive Error", err);
  } else {
    // Make sure we have a message
    if (data.Messages != null) {
      var visibilityParams = {
        QueueUrl: queueURL,
        ReceiptHandle: data.Messages[0].ReceiptHandle,
        VisibilityTimeout: 20, // 20 second timeout
      };
      sqs.changeMessageVisibility(visibilityParams, function (err, data) {
        if (err) {
          console.log("Delete Error", err);
        } else {
          console.log("Timeout Changed", data);
        }
      });
    } else {
      console.log("No messages to change");
    }
  }
```

```
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node sqs_changingvisibility.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

Amazon SQS에서 긴 폴링 활성화

이 Node.js 코드 예제는 다음을 보여 줍니다.

- 새로 생성된 대기열에 대해 긴 폴링을 활성화하는 방법
- 기존 대기열에 대해 긴 폴링을 활성화하는 방법
- 메시지를 수신할 때 긴 폴링을 활성화하는 방법.

시나리오

긴 폴링은 응답을 전송하기 전에 대기열에서 메시지를 사용할 수 있을 때까지 Amazon SQS를 지정된 시간 동안 대기시켜 빈 응답의 개수를 줄입니다. 또한, 긴 폴링은 서버의 샘플링 대신에 모든 서버를 쿼리하여 False인 빈 응답을 제거합니다. 긴 폴링을 활성화하려면 수신된 메시지에 0이 아닌 대기 시간을 지정해야 합니다. 이렇게 하려면 대기열의 `ReceiveMessageWaitTimeSeconds` 파라미터를 설정하거나 메시지가 수신되었을 때 메시지에서 `WaitTimeSeconds` 파라미터를 설정하면 됩니다.

이 예제에서는 일련의 Node.js 모듈을 사용하여 긴 폴링을 활성화합니다. Node.js 모듈은 SDK for JavaScript로 `AWS.SQS` 클라이언트 클래스의 다음 메서드를 사용하여 긴 폴링을 활성화합니다.

- `setQueueAttributes`
- `receiveMessage`
- `createQueue`

Amazon SQS 긴 폴링에 대한 자세한 내용은 Amazon Simple Queue Service 개발자 안내서의 긴 폴링을 참조하세요.

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 이러한 작업들을 완료해야 합니다.

- Node.js를 설치합니다. Node.js 설치에 대한 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 공유 자격 증명 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.

대기열 생성 시 긴 폴링 활성화

파일 이름이 `sqs_longpolling_createqueue.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. Amazon SQS에 액세스하려면 `AWS.SQS` 서비스 객체를 생성합니다. `ReceiveMessageWaitTimeSeconds` 파라미터의 0이 아닌 값을 포함하여 대기열을 생성하는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. `createQueue` 메서드를 호출합니다. 그러면 대기열에 대해 긴 폴링이 활성화됩니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the SQS service object
var sqs = new AWS.SQS({ apiVersion: "2012-11-05" });

var params = {
  QueueName: "SQS_QUEUE_NAME",
  Attributes: {
    ReceiveMessageWaitTimeSeconds: "20",
  },
};

sqs.createQueue(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.QueueUrl);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node sqs_longpolling_createqueue.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

기존 대기열에 대해 긴 폴링 활성화

파일 이름이 `sqs_longpolling_existingqueue.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. Amazon Simple Queue Service에 액세스하려면 `AWS.SQS` 서비스 객체를 생성합니다. `ReceiveMessageWaitTimeSeconds` 파라미터의 0이 아닌 값 및 대기열의 URL을 포함하여 대기열의 속성을 설정하는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. `setQueueAttributes` 메서드를 호출합니다. 그러면 대기열에 대해 긴 폴링이 활성화됩니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the SQS service object
var sqs = new AWS.SQS({ apiVersion: "2012-11-05" });

var params = {
  Attributes: {
    ReceiveMessageWaitTimeSeconds: "20",
  },
  QueueUrl: "SQS_QUEUE_URL",
};

sqs.setQueueAttributes(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node sqs_longpolling_existingqueue.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

메시지 수신 시 긴 폴링 활성화

파일 이름이 `sqs_longpolling_receivemessage.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. Amazon Simple Queue Service에 액세스하려면 `AWS.SQS` 서비스 객체를 생성합니다. `WaitTimeSeconds` 파라미터의 0이 아닌 값 및 대기열의 URL을 포함하여 메시지를 수신하는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. `receiveMessage` 메서드를 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the SQS service object
var sqs = new AWS.SQS({ apiVersion: "2012-11-05" });

var queueURL = "SQS_QUEUE_URL";

var params = {
  AttributeNames: ["SentTimestamp"],
  MaxNumberOfMessages: 1,
  MessageAttributeNames: ["All"],
  QueueUrl: queueURL,
  WaitTimeSeconds: 20,
};

sqs.receiveMessage(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node sqs_longpolling_receivemessage.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

Amazon SQS에서 DLQ(Dead Letter Queue) 사용

이 Node.js 코드 예제는 다음을 보여 줍니다.

- 대기열이 처리할 수 없는 다른 대기열의 메시지를 수신하고 보관하기 위해 대기열을 사용하는 방법

시나리오

배달 못한 편지 대기열은 성공적으로 처리하지 못한 메시지를 다른 (소스) 대기열에서 보낼 수 있는 대기열입니다. 배달 못한 편지 대기열에서 이 메시지를 구분하고 격리하여 처리에 실패한 이유를 확인할 수 있습니다. 배달 못한 편지 대기열로 메시지를 보내는 각 소스 대기열을 개별적으로 구성해야 합니다. 여러 대기열은 단일 배달 못한 편지 대기열을 대상으로 삼을 수 있습니다.

이 예제에서는 Node.js 모듈을 사용하여 배달 못한 편지 대기열로 메시지를 라우팅합니다. Node.js 모듈은 SDK for JavaScript로 `AWS.SQS` 클라이언트 클래스의 다음 메서드를 사용하여 DLQ(Dead Letter Queue)를 사용합니다.

- `setQueueAttributes`

Amazon SQS DLQ(Dead Letter Queue)에 대한 자세한 내용은 Amazon Simple Queue Service 개발자 안내서의 Amazon SQS DLQ(Dead Letter Queue) 사용을 참조하세요.

사전 필수 작업

이 예제를 설정하고 실행하려면 먼저 이러한 작업들을 완료해야 합니다.

- Node.js를 설치합니다. Node.js 설치에 대한 자세한 내용은 Node.js 웹 사이트를 참조하세요.
- 사용자 자격 증명을 사용하여 공유 구성 파일을 생성합니다. 공유 자격 증명 파일 제공에 대한 자세한 내용은 공유 인증 자격 증명 파일에서 Node.js에 인증 자격 증명 로드 섹션을 참조하세요.
- DLQ(Dead Letter Queue)로 제공할 Amazon SQS 대기열을 생성합니다. 대기열 생성에 대한 예제는 Amazon SQS에서 대기열 사용 섹션을 참조하세요.

소스 대기열 구성

배달 못한 편지 대기열 역할을 하는 대기열을 생성한 후에는 다른 대기열이 처리되지 못한 메시지를 배달 못한 편지 대기열로 라우팅하도록 구성해야 합니다. 이렇게 하려면 배달 못한 편지 대기열로 사용할 대기열을 식별하는 리드라이브 정책을 지정하고 개별 메시지가 배달 못한 편지 대기열로 라우팅되기 전 최대 수신 수를 설정해야 합니다.

파일 이름이 `sqs_deadletterqueue.js`인 Node.js 모듈을 생성합니다. 위와 같이 SDK를 구성해야 합니다. Amazon SQS에 액세스하려면 `AWS.SQS` 서비스 객체를 생성합니다. 배달 못한 편지 대기열의 ARN과 `maxReceiveCount`의 값을 모두 지정하는 `RedrivePolicy` 파라미터를 포함하여 대기열 속성을 업데이트하는 데 필요한 파라미터를 포함하는 JSON 객체를 생성합니다. 또한 구성하고자 하는 URL 소스 대기열을 지정합니다. `setQueueAttributes` 메서드를 호출합니다.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
AWS.config.update({ region: "REGION" });

// Create the SQS service object
var sqs = new AWS.SQS({ apiVersion: "2012-11-05" });

var params = {
  Attributes: {
    RedrivePolicy:
      '{"deadLetterTargetArn":"DEAD_LETTER_QUEUE_ARN","maxReceiveCount":"10"}',
  },
  QueueUrl: "SOURCE_QUEUE_URL",
};

sqs.setQueueAttributes(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

예제를 실행하려면 명령줄에서 다음을 입력합니다.

```
node sqs_deadletterqueue.js
```

이 샘플 코드는 GitHub에서 찾을 수 있습니다.

자습서

다음 자습서에서는 AWS SDK for JavaScript 사용과 관련하여 다양한 작업을 수행하는 방법을 보여 줍니다.

주제

- 자습서: Amazon EC2 인스턴스에서 Node.js 설정

자습서: Amazon EC2 인스턴스에서 Node.js 설정

SDK for JavaScript와 함께 Node.js를 사용하는 일반적인 시나리오는 Amazon Elastic Compute Cloud(Amazon EC2) 인스턴스에서 Node.js 웹 애플리케이션을 설정하고 실행하는 것입니다. 이 자습서에서는 Linux 인스턴스를 생성하고, SSH를 사용하여 해당 인스턴스에 연결한 다음, 해당 인스턴스에서 실행할 Node.js를 설치합니다.

사전 조건

이 자습서에서는 인터넷에서 접근 가능하고 SSH를 사용하여 연결할 수 있으며 퍼블릭 DNS 이름이 있는 Linux 인스턴스를 이미 시작했다고 가정합니다. 자세한 내용은 Amazon EC2 사용 설명서의 1단계: 인스턴스 시작을 참조하세요.

Important

새 Amazon EC2 인스턴스를 시작할 때 Amazon Linux 2023 Amazon Machine Image(AMI)를 사용합니다.

보안 그룹이 SSH(포트 22), HTTP(포트 80), HTTPS(포트 443) 연결을 허용하도록 구성되어야 합니다. 이러한 사전 조건에 관한 자세한 내용은 Amazon EC2 사용 설명서의 Amazon EC2 사용 설정 섹션을 참조하세요.

절차

다음 절차는 Amazon Linux 인스턴스에서 Node.js를 설치하는 데 도움이 됩니다. 이 서버를 사용하여 Node.js 웹 애플리케이션을 호스팅할 수 있습니다.

Linux 인스턴스에서 Node.js를 설정하려면

1. SSH를 사용하여 `ec2-user`로 Linux 인스턴스에 연결합니다.
2. 명령줄에 다음을 입력하여 nvm(노드 버전 관리자)을 설치합니다.

   > **Warning**
   >
   > AWS는 다음 코드를 제어하지 않습니다. 실행하기 전에 먼저 신뢰성과 무결성을 확인해야 합니다. 이 코드에 대한 자세한 내용은 nvm GitHub 리포지토리에서 확인할 수 있습니다.

   ```
   curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.7/install.sh | bash
   ```

   nvm을 사용하면 여러 버전의 Node.js를 설치할 수 있고 여러 버전 간을 전환할 수 있기 때문에 여기서는 nvm을 사용하여 Node.js를 설치합니다.
3. 명령줄에 다음을 입력하여 nvm을 로드합니다.

   ```
   source ~/.bashrc
   ```
4. 명령줄에 다음을 입력하여 nvm을 사용해 최신 LTS 버전의 Node.js를 설치합니다.

   ```
   nvm install --lts
   ```

   Node.js를 설치하면 npm(노드 패키지 관리자)도 설치되므로 필요에 따라 추가 모듈을 설치할 수 있습니다.
5. 명령줄에 다음을 입력하여 Node.js가 올바르게 설치되고 실행되는지 테스트합니다.

   ```
   node -e "console.log('Running Node.js ' + process.version)"
   ```

   이렇게 하면 실행 중인 Node.js의 버전을 보여 주는 메시지가 다음과 같이 표시됩니다.

   `Running Node.js` *`VERSION`*

> **Note**
>
> 노드 설치는 현재 Amazon EC2 세션에만 적용됩니다. CLI 세션을 다시 시작하는 경우 nvm을 사용하여 설치된 노드 버전을 활성화해야 합니다. 인스턴스가 종료되면 노드를 다시 설치해야

합니다. 다음 섹션에 설명된 대로 유지하려는 구성이 있는 경우 대안은 Amazon EC2 인스턴스의 Amazon Machine Image(AMI)를 만드는 것입니다.

Amazon Machine Image 생성

Amazon EC2 인스턴스에 Node.js를 설치한 후에는 해당 인스턴스에서 Amazon Machine Image(AMI)를 생성할 수 있습니다. AMI를 생성하면 동일한 Node.js 설치에서 여러 Amazon EC2 인스턴스를 쉽게 프로비저닝할 수 있습니다. 기존 인스턴스에서 AMI를 생성하는 방법에 관한 자세한 내용은 Amazon EC2 사용 설명서의 Amazon EBS-backed Linux AMI 생성을 참조하세요.

관련 리소스

이 주제에서 사용되는 명령과 소프트웨어에 대한 자세한 내용은 다음 웹 페이지를 참조하세요.

- nvm(노드 버전 관리자): GitHub의 nvm 리포지토리를 참조하세요.
- npm(노드 패키지 관리자): npm 웹 사이트를 참조하세요.

JavaScript API 참조

SDK for JavaScript 최신 버전에 대한 API 참조 항목 위치:

AWS SDK for JavaScript API 참조 안내서

GitHub의 SDK 변경 로그

버전 2.4.8 이상 릴리스의 변경 로그 위치:

변경 로그

AWS SDK for JavaScript의 v3로 마이그레이션

AWS SDK for JavaScript 버전 3은 버전 2의 메이저 재작성 버전입니다. 버전 3으로 마이그레이션에 관한 자세한 내용은 AWS SDK for JavaScript 개발자 안내서 v3의 Migrate from version 2.x to 3.x of the AWS SDK for JavaScript 섹션을 참조하시기 바랍니다.

이 AWS 제품 또는 서비스에 대한 보안

Amazon Web Services(AWS)에서 가장 우선순위가 높은 것이 클라우드 보안입니다. AWS 고객으로서 여러분은 가장 높은 보안 요구 사항을 충족하기 위해 설계된 데이터 센터 및 네트워크 아키텍처의 혜택을 받게 됩니다. 보안은 AWS과 사용자의 공동 책임입니다. 공동 책임 모델은 이 사항을 클라우드 내 보안 및 클라우드의 보안으로 설명합니다.

클라우드의 보안 – AWS는 AWS 클라우드에서 모든 서비스를 실행하는 인프라를 보호하며 안전하게 사용할 수 있는 서비스를 제공합니다. 당사의 보안 책임은 AWS에서 우선 순위가 가장 높으며, 타사 감사자는 AWS 규정 준수 프로그램의 일환으로 정기적으로 보안 효율성을 테스트하고 검증합니다.

클라우드 내 보안 – 사용자의 책임은 사용하는 AWS 서비스, 그리고 데이터의 민감도, 조직의 요구 사항, 관련 법률 및 규정을 비롯한 기타 요소에 의해 결정됩니다.

이 AWS 제품 또는 서비스는 지원하는 특정 Amazon Web Services(AWS) 서비스를 통해 공동 책임 모델을 따릅니다. AWS 서비스 보안 정보는 AWS 서비스 보안 설명서 페이지 및 AWS 규정 준수 프로그램의 AWS 규정 준수 업무 범위에 속하는 서비스를 참조하세요.

주제

- 이 AWS 제품 또는 서비스의 데이터 보호
- 자격 증명 및 액세스 관리
- 이 AWS 제품 또는 서비스에 대한 규정 준수 확인
- 이 AWS 제품 또는 서비스에 대한 복원력
- 이 AWS 제품 또는 서비스에 대한 인프라 보안
- TLS의 최소 버전 적용

이 AWS 제품 또는 서비스의 데이터 보호

AWS 공동 책임 모델은 이 AWS 제품 또는 서비스의 데이터 보호에 적용됩니다. 이 모델에서 설명하는 것처럼 AWS는 모든 AWS 클라우드를 실행하는 글로벌 인프라를 보호할 책임이 있습니다. 사용자는 이 인프라에 호스팅되는 콘텐츠에 대한 통제 권한을 유지할 책임이 있습니다. 사용하는 AWS 서비스의 보안 구성과 관리 태스크에 대한 책임도 사용자에게 있습니다. 데이터 프라이버시에 관한 자세한 내용은 데이터 프라이버시 FAQ를 참조하세요. 유럽의 데이터 보호에 대한 자세한 내용은 AWS보안 블로그의 AWS공동 책임 모델 및 GDPR 블로그 게시물을 참조하세요.

데이터를 보호하려면 AWS 계정자격 증명을 보호하고 AWS IAM Identity Center또는 AWS Identity and Access Management(IAM)를 통해 개별 사용자 계정을 설정하는 것이 좋습니다. 이렇게 하면 개별 사용자에게 자신의 직무를 충실히 이행하는 데 필요한 권한만 부여됩니다. 또한 다음과 같은 방법으로 데이터를 보호하는 것이 좋습니다.

- 각 계정에 다중 인증(MFA)을 사용합니다.
- SSL/TLS를 사용하여 AWS리소스와 통신하세요. TLS 1.2는 필수이며 TLS 1.3을 권장합니다.
- AWS CloudTrail으로 API 및 사용자 활동 로깅을 설정하세요. AWS 활동 캡처에 CloudTrail 추적을 사용하는 방법에 대한 자세한 내용은 AWS CloudTrail사용 설명서의 CloudTrail 추적 작업을 참조하세요.
- AWS 암호화 솔루션을 AWS 서비스내의 모든 기본 보안 컨트롤과 함께 사용하세요.
- Amazon S3에 저장된 민감한 데이터를 검색하고 보호하는 데 도움이 되는 Amazon Macie와 같은 고급 관리형 보안 서비스를 사용합니다.
- 명령줄 인터페이스 또는 API를 통해 AWS에 액세스할 때 FIPS 140-3 검증된 암호화 모듈이 필요한 경우, FIPS 엔드포인트를 사용합니다. 사용 가능한 FIPS 엔드포인트에 대한 자세한 내용은 연방 정보 처리 표준(FIPS) 140-3을 참조하세요.

고객의 이메일 주소와 같은 기밀 정보나 중요한 정보는 태그나 이름 필드와 같은 자유 형식 텍스트 필드에 입력하지 않는 것이 좋습니다. 여기에는 콘솔, API, AWS CLI 또는 AWS SDK를 사용하여 이 AWS 제품이나 서비스 또는 기타 AWS 서비스 서비스로 작업하는 경우도 포함됩니다. 이름에 사용되는 태그 또는 자유 형식 텍스트 필드에 입력하는 모든 데이터는 청구 또는 진단 로그에 사용될 수 있습니다. 외부 서버에 URL을 제공할 때 해당 서버에 대한 요청을 검증하기 위해 보안 인증 정보를 URL에 포함시켜서는 안 됩니다.

자격 증명 및 액세스 관리

AWS Identity and Access Management(IAM)는 관리자가 AWS 리소스에 대한 액세스를 안전하게 제어할 수 있도록 지원하는 AWS 서비스입니다. IAM 관리자는 어떤 사용자가 AWS 리소스를 사용할 수 있는 인증(로그인) 및 권한(권한 있음)을 받을 수 있는지 제어합니다. IAM은 추가 비용 없이 사용할 수 있는 AWS 서비스입니다.

주제

- 고객
- 보안 인증을 통한 인증
- 정책을 사용하여 액세스 관리

- AWS 서비스에서 IAM을 사용하는 방식
- AWS 보안 인증 및 액세스 문제 해결

고객

AWS Identity and Access Management(IAM)을 사용하는 방법은 AWS에서 수행하는 작업에 따라 달라집니다.

서비스 사용자 – AWS 서비스를 사용하여 작업을 수행하는 경우 필요한 보안 인증 정보와 권한을 관리자가 제공합니다. 더 많은 AWS 기능을 사용하여 작업을 수행하게 되면 추가 권한이 필요할 수 있습니다. 액세스 권한 관리 방법을 이해하면 관리자에게 올바른 권한을 요청하는 데 도움이 됩니다. AWS의 기능에 액세스할 수 없는 경우 AWS 보안 인증 및 액세스 문제 해결 또는 사용 중인 AWS 서비스의 사용 설명서를 참조하세요.

서비스 관리자 - 회사에서 AWS 리소스를 책임지고 있는 담당자라면 AWS에 대한 전체 액세스 권한을 가지고 있을 것입니다. 서비스 관리자는 서비스 사용자가 액세스해야 하는 AWS 기능과 리소스를 결정합니다. 그런 다음, IAM 관리자에게 요청을 제출하여 서비스 사용자의 권한을 변경해야 합니다. 이 페이지의 정보를 검토하여 IAM의 기본 개념을 이해하세요. 회사에서 AWS와 함께 IAM을 사용하는 방법에 대한 자세한 내용은 사용 중인 AWS 서비스의 사용 설명서를 참조하세요.

IAM 관리자 - IAM 관리자라면 AWS에 대한 액세스 권한 관리 정책 작성 방법을 자세히 알고 싶을 것입니다. IAM에서 사용할 수 있는 AWS 보안 인증 기반 정책 예제를 보려면 사용 중인 AWS 서비스의 사용 설명서를 참조하세요.

보안 인증을 통한 인증

인증은 ID 자격 증명을 사용하여 AWS에 로그인하는 방식입니다. AWS 계정 루트 사용자이나 IAM 사용자로, 또는 IAM 역할을 수임하여 인증(에 로그인)받아야 합니다.

AWS IAM Identity Center(IAM Identity Center), Single Sign-On 인증 또는 Google/Facebook 자격 증명과 같은 자격 증명 소스의 자격 증명을 사용하여 페더레이션 ID로 로그인할 수 있습니다. 로그인하는 방법에 대한 자세한 내용은 AWS 로그인사용 설명서의 AWS 계정에 로그인하는 방법 섹션을 참조하세요.

프로그래밍 방식 액세스를 위해 AWS는 요청에 암호화 방식으로 서명할 수 있는 SDK 및 CLI를 제공합니다. 자세한 내용은 IAM 사용 설명서의 API 요청용 AWS Signature Version 4 섹션을 참조하세요.

AWS 계정 루트 사용자

AWS 계정을 생성하는 경우에는 모든 AWS 서비스 서비스와 리소스에 대한 완전한 액세스 권한이 있는 AWS 계정 루트 사용자라는 단일 로그인 ID로 시작합니다. 일상적인 태스크에 루트 사용자를 사용하지 않을 것을 강력히 권장합니다. 루트 사용자 자격 증명이 필요한 작업은 IAM 사용 설명서의 [루트 사용자 자격 증명이 필요한 작업](#) 섹션을 참조하세요.

페더레이션 ID

가장 좋은 방법은 인간 사용자가 ID 공급자와의 페더레이션을 사용하여 임시 자격 증명으로 AWS 서비스에 액세스하도록 하는 것입니다.

페더레이션 ID는 엔터프라이즈 사용자 디렉터리, 웹 ID 제공업체 또는 Directory Service의 사용자로, ID의 자격 증명을 사용하여 AWS 서비스에 액세스합니다. 페더레이션 ID는 임시 자격 증명을 제공하는 역할을 수임합니다.

중앙 집중식 액세스 관리를 위해 AWS IAM Identity Center를 추천합니다. 자세한 정보는 AWS IAM Identity Center사용 설명서의 [What is IAM Identity Center?](#)를 참조하세요.

IAM 사용자 및 그룹

[IAM 사용자](#)는 단일 개인 또는 애플리케이션에 대한 특정 권한을 가진 ID입니다. 장기 자격 증명이 있는 IAM 사용자 대신 임시 자격 증명을 사용하는 것이 좋습니다. 자세한 내용은 IAM 사용 설명서에서 [임시 자격 증명을 사용하여 AWS에 액세스하려면 인간 사용자가 ID 제공업체와의 페더레이션을 사용하도록 요구](#)를 참조하세요.

[IAM 그룹](#)은 IAM 사용자 모음을 지정하고 대규모 사용자 집합에 대한 관리 권한을 더 쉽게 만듭니다. 자세한 내용은 IAM 사용 설명서의 [IAM 사용자 사용 사례](#) 섹션을 참조하세요.

IAM 역할

[IAM 역할](#)은 임시 자격 증명을 제공하는 특정 권한이 있는 자격 증명입니다. [사용자에서 IAM 역할(콘솔)로 전환](#)하거나 AWS CLI 또는 AWS API 작업을 직접적으로 호출하여 역할을 수임할 수 있습니다. 자세한 내용은 IAM 사용 설명서의 [역할 수임 방법](#)을 참조하세요.

IAM 역할은 페더레이션 사용자 액세스, 임시 IAM 사용자 권한, 교차 계정 액세스, 교차 서비스 액세스 및 Amazon EC2에서 실행되는 애플리케이션에 유용합니다. 자세한 내용은 IAM 사용 설명서의 [교차 계정 리소스 액세스](#)를 참조하세요.

정책을 사용하여 액세스 관리

정책을 생성하고 AWS ID 또는 리소스에 연결하여 AWS에서 내 액세스를 제어합니다. 정책은 자격 증명이나 리소스와 연결될 때 해당 권한을 정의합니다. AWS는 보안 주체가 요청을 보낼 때 이러한 정책을 평가합니다. 대부분의 정책은 AWS에 JSON 문서로 저장됩니다. JSON 정책 문서에 대한 자세한 내용은 IAM 사용 설명서의 JSON 정책 개요 섹션을 참조하세요.

정책을 사용하여 관리자는 어떤 보안 주체가 어떤 리소스에 대해 어떤 조건에서 작업을 수행할 수 있는지 정의하여 누가 무엇을 액세스할 수 있는지 지정합니다.

기본적으로 사용자 및 역할에는 어떠한 권한도 없습니다. IAM 관리자는 IAM 정책을 생성하고 사용자가 수임할 수 있는 역할에 추가합니다. IAM 정책은 작업을 수행하기 위해 사용하는 방법과 관계없이 작업에 대한 권한을 정의합니다.

ID 기반 정책

ID 기반 정책은 ID(사용자, 사용자 그룹 또는 역할)에 연결하는 JSON 권한 정책 문서입니다. 이러한 정책은 자격 증명이 수행할 수 있는 작업, 대상 리소스 및 이에 관한 조건을 제어합니다. ID 기반 정책을 생성하는 방법을 알아보려면 IAM 사용 설명서에서 고객 관리형 정책으로 사용자 지정 IAM 권한 정의를 참조하세요.

ID 기반 정책은 인라인 정책(단일 ID에 직접 포함) 또는 관리형 정책(여러 ID에 연결된 독립 실행형 정책)일 수 있습니다. 관리형 정책 또는 인라인 정책을 선택하는 방법을 알아보려면 IAM 사용 설명서의 관리형 정책 및 인라인 정책 중에서 선택 섹션을 참조하세요.

리소스 기반 정책

리소스 기반 정책은 리소스에 연결하는 JSON 정책 설명서입니다. 예를 들어 IAM 역할 신뢰 정책 및 Amazon S3 버킷 정책이 있습니다. 리소스 기반 정책을 지원하는 서비스에서 서비스 관리자는 이러한 정책을 사용하여 특정 리소스에 대한 액세스를 통제할 수 있습니다. 리소스 기반 정책에서 보안 주체를 지정해야 합니다.

리소스 기반 정책은 해당 서비스에 있는 인라인 정책입니다. 리소스 기반 정책에서는 IAM의 AWS 관리형 정책을 사용할 수 없습니다.

액세스 제어 목록(ACL)

액세스 제어 목록(ACL)은 어떤 위탁자(계정 멤버, 사용자 또는 역할)가 리소스에 액세스할 수 있는 권한을 가지고 있는지를 제어합니다. ACLs는 JSON 정책 문서 형식을 사용하지 않지만 리소스 기반 정책과 유사합니다.

Amazon S3, AWS WAF 및 Amazon VPC는 ACL을 지원하는 대표적인 서비스입니다. ACL에 관한 자세한 내용은 Amazon Simple Storage Service 개발자 가이드의 액세스 제어 목록(ACL) 개요를 참조하세요.

기타 정책 유형

AWS는 이러한 정책 타입이 부여하는 최대 권한을 설정할 수 있는 추가 정책 타입을 지지합니다.

- 권한 경계 - ID 기반 정책에서 IAM 엔터티에 부여할 수 있는 최대 권한을 설정합니다. 자세한 정보는 IAM 사용 설명서의 IAM 엔터티의 권한 범위를 참조하세요.
- 서비스 제어 정책(SCP) - AWS Organizations내 조직 또는 조직 단위에 대한 최대 권한을 지정합니다. 자세한 내용은 AWS Organizations사용 설명서의 서비스 제어 정책을 참조하세요.
- 리소스 제어 정책(RCP) – 계정의 리소스에 사용할 수 있는 최대 권한을 설정합니다. 자세한 내용은 AWS Organizations사용 설명서의 리소스 제어 정책(RCP)을 참조하세요.
- 세션 정책 – 역할 또는 페더레이션 사용자에 대해 임시 세션을 프로그래밍 방식으로 생성할 때 파라미터로 전달하는 고급 정책입니다. 자세한 내용은 IAM 사용 설명서의 세션 정책을 참조하세요.

여러 정책 유형

여러 정책 유형이 요청에 적용되는 경우, 결과 권한은 이해하기가 더 복잡합니다. 여러 정책 유형이 관련될 때 AWS가 요청을 허용할지를 결정하는 방법을 알아보려면 IAM 사용 설명서의 정책 평가 로직을 참조하세요.

AWS 서비스에서 IAM을 사용하는 방식

AWS 서비스에서 대부분의 IAM 기능을 사용하는 방법을 전체적으로 알아보려면 IAM 사용 설명서의 IAM으로 작업하는 AWS 서비스를 참조하세요.

특정 AWS 서비스에 IAM을 사용하는 방법을 알아보려면 관련 서비스 사용 설명서의 보안 섹션을 참조하세요.

AWS 보안 인증 및 액세스 문제 해결

다음 정보를 사용하여 AWS 및 IAM에서 발생할 수 있는 공통적인 문제를 진단하고 수정할 수 있습니다.

주제

- AWS에서 작업을 수행할 권한이 없음
- iam:PassRole을 수행하도록 인증되지 않음

- 내 AWS 계정 외부의 사람이 내 AWS 리소스에 액세스할 수 있게 허용하기를 원합니다.

AWS에서 작업을 수행할 권한이 없음

작업을 수행할 권한이 없다는 오류가 표시되면 작업을 수행할 수 있도록 정책을 업데이트해야 합니다.

다음의 예제 오류는 `mateojackson` IAM 사용자가 콘솔을 사용하여 가상 *my-example-widget* 리소스에 대한 세부 정보를 보려고 하지만 가상 `awes:GetWidget` 권한이 없을 때 발생합니다.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 awes:GetWidget on resource: my-example-widget
```

이 경우, `awes:GetWidget` 작업을 사용하여 *my-example-widget* 리소스에 액세스할 수 있도록 `mateojackson` 사용자 정책을 업데이트해야 합니다.

도움이 필요한 경우 AWS 관리자에게 문의하세요. 관리자는 로그인 자격 증명을 제공한 사람입니다.

iam:PassRole을 수행하도록 인증되지 않음

`iam:PassRole` 작업을 수행할 수 있는 권한이 없다는 오류가 수신되면 AWS에 역할을 전달할 수 있도록 정책을 업데이트해야 합니다.

일부 AWS 서비스에서는 새 서비스 역할 또는 서비스 연결 역할을 생성하는 대신, 해당 서비스에 기존 역할을 전달할 수 있습니다. 이렇게 하려면 사용자가 서비스에 역할을 전달할 수 있는 권한을 가지고 있어야 합니다.

다음 예 오류는 `marymajor`라는 IAM 사용자가 콘솔을 사용하여 AWS에서 작업을 수행하려고 하는 경우에 발생합니다. 하지만 작업을 수행하려면 서비스 역할이 부여한 권한이 서비스에 있어야 합니다. Mary는 서비스에 역할을 전달할 수 있는 권한을 가지고 있지 않습니다.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

이 경우, Mary가 `iam:PassRole`작업을 수행할 수 있도록 Mary의 정책을 업데이트해야 합니다.

도움이 필요한 경우 AWS 관리자에게 문의하세요. 관리자는 로그인 자격 증명을 제공한 사람입니다.

내 AWS 계정 외부의 사람이 내 AWS 리소스에 액세스할 수 있게 허용하기를 원합니다.

다른 계정의 사용자 또는 조직 외부의 사람이 리소스에 액세스할 때 사용할 수 있는 역할을 생성할 수 있습니다. 역할을 수임할 신뢰할 수 있는 사람을 지정할 수 있습니다. 리소스 기반 정책 또는 액세스 제

어 목록(ACL)을 지원하는 서비스의 경우, 이러한 정책을 사용하여 다른 사람에게 리소스에 대한 액세스 권한을 부여할 수 있습니다.

자세히 알아보려면 다음을 참조하세요.

- AWS에서 이러한 기능을 지원하는지 여부를 알아보려면 AWS 서비스에서 IAM을 사용하는 방식을 참조하세요.
- 소유하고 있는 AWS 계정의 리소스에 대한 액세스 권한을 제공하는 방법을 알아보려면 IAM 사용자 설명서의 자신이 소유한 다른 AWS 계정의 IAM 사용자에 대한 액세스 권한 제공을 참조하세요.
- 리소스에 대한 액세스 권한을 서드 파티 AWS 계정에게 제공하는 방법을 알아보려면 IAM 사용 설명서의 서드 파티가 소유한 AWS 계정에 대한 액세스 제공을 참조하세요.
- ID 페더레이션을 통해 액세스 권한을 제공하는 방법을 알아보려면 IAM 사용 설명서의 외부에서 인증된 사용자에게 액세스 권한 제공(ID 페더레이션)을 참조하세요.
- 크로스 계정 액세스에 대한 역할과 리소스 기반 정책 사용의 차이점을 알아보려면 IAM 사용 설명서의 IAM의 크로스 계정 리소스 액세스를 참조하세요.

이 AWS 제품 또는 서비스에 대한 규정 준수 확인

AWS 서비스가 특정 규정 준수 프로그램의 범위에 포함되는지 알아보려면 규정 준수 프로그램 제공 범위 내 AWS 서비스를 참조하고 관심 있는 규정 준수 프로그램을 선택하세요. 일반적인 정보는 AWS 규정 준수 프로그램을 참조하세요.

AWS Artifact를 사용하여 타사 감사 보고서를 다운로드할 수 있습니다. 자세한 내용은 AWS Artifact에서 보고서 다운로드를 참조하세요.

AWS 서비스 사용 시 규정 준수 책임은 데이터의 민감도, 회사의 규정 준수 목표 및 관련 법률 및 규정에 따라 결정됩니다. AWS 서비스 사용 시 규정 준수 책임에 대한 자세한 내용은 AWS 보안 설명서를 참조하세요.

이 AWS 제품 또는 서비스는 지원하는 특정 Amazon Web Services(AWS) 서비스를 통해 공동 책임 모델을 따릅니다. AWS 서비스 보안 정보는 AWS 서비스 보안 설명서 페이지 및 AWS 규정 준수 프로그램의 AWS 규정 준수 업무 범위에 속하는 서비스를 참조하세요.

이 AWS 제품 또는 서비스에 대한 복원력

AWS 글로벌 인프라는 AWS 리전 및 가용 영역을 중심으로 구축됩니다.

AWS 리전에서는 물리적으로 분리되고 격리된 다수의 가용 영역을 제공하며 이러한 가용 영역은 짧은 지연 시간, 높은 처리량 및 높은 중복성을 갖춘 네트워크에 연결되어 있습니다.

가용 영역을 사용하면 중단 없이 영역 간에 자동으로 장애 극복 조치가 이루어지는 애플리케이션 및 데이터베이스를 설계하고 운영할 수 있습니다. 가용 영역은 기존의 단일 또는 다중 데이터 센터 인프라보다 가용성, 내결함성, 확장성이 뛰어납니다.

AWS 리전 및 가용 영역에 대한 자세한 내용은 AWS 글로벌 인프라를 참조하세요.

이 AWS 제품 또는 서비스는 지원하는 특정 Amazon Web Services(AWS) 서비스를 통해 공동 책임 모델을 따릅니다. AWS 서비스 보안 정보는 AWS 서비스 보안 설명서 페이지 및 AWS 규정 준수 프로그램의 AWS 규정 준수 업무 범위에 속하는 서비스를 참조하세요.

이 AWS 제품 또는 서비스에 대한 인프라 보안

이 AWS 제품 또는 서비스는 관리형 서비스를 사용하므로 AWS 글로벌 네트워크 보안으로 보호됩니다. AWS 보안 서비스와 AWS의 인프라 보호 방법에 대한 자세한 내용은 AWS 클라우드 보안을 참조하세요. 인프라 보안에 대한 모범 사례를 사용하여 AWS 환경을 설계하려면 보안 원칙 AWS Well-Architected 프레임워크의 인프라 보호를 참조하세요.

AWS에서 게시한 API 직접 호출을 사용하여 네트워크를 통해 이 AWS 제품 또는 서비스에 액세스합니다. 클라이언트는 다음을 지원해야 합니다.

- Transport Layer Security(TLS). TLS 1.2는 필수이며 TLS 1.3을 권장합니다.
- DHE(Ephemeral Diffie-Hellman) 또는 ECDHE(Elliptic Curve Ephemeral Diffie-Hellman)와 같은 완전 전송 보안(PFS)이 포함된 암호 제품군. Java 7 이상의 최신 시스템은 대부분 이러한 모드를 지원합니다.

또한 요청은 액세스 키 ID 및 IAM 위탁자와 관련된 시크릿 액세스 키를 사용하여 서명해야 합니다. 또는 AWS Security Token Service(AWS STS)를 사용하여 임시 보안 보안 인증을 생성하여 요청에 서명할 수 있습니다.

이 AWS 제품 또는 서비스는 지원하는 특정 Amazon Web Services(AWS) 서비스를 통해 공동 책임 모델을 따릅니다. AWS 서비스 보안 정보는 AWS 서비스 보안 설명서 페이지 및 AWS 규정 준수 프로그램의 AWS 규정 준수 업무 범위에 속하는 서비스를 참조하세요.

TLS의 최소 버전 적용

AWS 서비스와 통신할 때 보안을 강화하려면 TLS 1.2 이상을 사용하도록 AWS SDK for JavaScript를 구성합니다.

전송 계층 보안(TLS)은 웹 브라우저 및 기타 애플리케이션에서 네트워크를 통해 교환되는 데이터의 프라이버시 및 무결성을 보장하기 위해 사용하는 프로토콜입니다.

Important

2024년 6월 10일부터 각 AWS 리전의 AWS 서비스 API 엔드포인트에서 TLS 1.3을 사용할 수 있게 되었다고 발표했습니다. AWS SDK for JavaScript v2는 TLS 버전 자체를 협상하지 않습니다. 대신, `https.Agent`를 통해 구성할 수 있는 Node.js에서 결정된 TLS 버전을 사용합니다. AWS에서는 Node.js의 현재 활성 LTS 버전을 사용할 것을 권장합니다.

Node.js에서 TLS 확인 및 적용

AWS SDK for JavaScript를 Node.js와 함께 사용하면 기본 Node.js 보안 계층을 사용하여 TLS 버전을 설정합니다.

Node.js 12.0.0 이상에서는 TLS 1.3을 지원하는 OpenSSL 1.1.1b의 최소 버전을 사용합니다. AWS SDK for JavaScript v2에서는 사용 가능한 경우 기본적으로 TLS 1.3을 사용하지만, 필요한 경우 기본적으로 더 낮은 버전을 사용합니다.

OpenSSL 및 TLS의 버전 확인

컴퓨터에 Node.js에서 사용하는 OpenSSL의 버전을 얻으려면 다음 명령을 실행합니다.

```
node -p process.versions
```

목록에 있는 OpenSSL 버전은 다음 예제와 같이 Node.js에서 사용하는 버전입니다.

```
openssl: '1.1.1b'
```

컴퓨터에서 Node.js에서 사용하는 TLS 버전을 얻으려면 노드 셸을 시작하고 순서대로 다음 명령을 실행합니다.

```
> var tls = require("tls");
```

```
> var tlsSocket = new tls.TLSSocket();
> tlsSocket.getProtocol();
```

마지막 명령은 다음 예제와 같이 TLS 버전을 출력합니다.

```
'TLSv1.3'
```

Node.js는 기본적으로 이 버전의 TLS를 사용하고 호출이 성공하지 못하면 다른 버전의 TLS를 협상하려고 시도합니다.

지원되는 최소 및 최대 TLS 버전 확인

개발자는 다음 스크립트를 사용하여 Node.js에서 지원되는 최소 및 최대 TLS 버전을 확인할 수 있습니다.

```
var tls = require("tls");
console.log("Supported TLS versions:", tls.DEFAULT_MIN_VERSION + " to " +
 tls.DEFAULT_MAX_VERSION);
```

마지막 명령은 다음 예제와 같이 기본값 최소 및 최대 TLS 버전을 출력합니다.

```
Supported TLS versions: TLSv1.2 to TLSv1.3
```

TLS의 최소 버전 적용

Node.js는 호출이 실패하면 TLS 버전을 협상합니다. 명령줄에서 스크립트를 실행할 때 또는 JavaScript 코드의 요청에 따라 이 협상 중에 허용 가능한 최소 TLS 버전을 적용할 수 있습니다.

명령줄에서 최소 TLS 버전을 지정하려면 Node.js 버전 11.4.0 이상을 사용해야 합니다. 특정 Node.js 버전을 설치하려면 먼저 노드 버전 관리자 설치 및 업데이트에 있는 단계를 사용하여 노드 버전 관리자(nvm)를 설치합니다. 그런 다음, 다음 명령을 실행하여 특정 버전의 Node.js를 설치하고 사용합니다.

```
nvm install 11
nvm use 11
```

Enforcing TLS 1.2

TLS 1.2가 허용 가능한 최소 버전인 경우 이를 적용하려면 다음 예제와 같이 스크립트를 실행할 때 `--tls-min-v1.2` 인수를 지정합니다.

```
node --tls-min-v1.2 yourScript.js
```

JavaScript 코드에서 특정 요청에 대해 허용 가능한 최소 TLS 버전을 지정하려면 다음 예제와 같이 `httpOptions` 파라미터를 사용하여 프로토콜을 지정합니다.

```
const https = require("https");
const {NodeHttpHandler} = require("@aws-sdk/node-http-handler");
const {DynamoDBClient} = require("@aws-sdk/client-dynamodb");

const client = new DynamoDBClient({
    region: "us-west-2",
    requestHandler: new NodeHttpHandler({
        httpsAgent: new https.Agent(
            {
                secureProtocol: 'TLSv1_2_method'
            }
        )
    })
});
```

Enforcing TLS 1.3

TLS 1.3이 허용 가능한 최소 버전인 경우 이를 적용하려면 다음 예와 같이 스크립트를 실행할 때 `--tls-min-v1.3` 인수를 지정합니다.

```
node --tls-min-v1.3 yourScript.js
```

JavaScript 코드에서 특정 요청에 대해 허용 가능한 최소 TLS 버전을 지정하려면 다음 예제와 같이 `httpOptions` 파라미터를 사용하여 프로토콜을 지정합니다.

```
const https = require("https");
const {NodeHttpHandler} = require("@aws-sdk/node-http-handler");
const {DynamoDBClient} = require("@aws-sdk/client-dynamodb");

const client = new DynamoDBClient({
    region: "us-west-2",
    requestHandler: new NodeHttpHandler({
        httpsAgent: new https.Agent(
            {
                secureProtocol: 'TLSv1_3_method'
            }
```

```
        )
    })
});
```

브라우저 스크립트에서 TLS 확인 및 적용

브라우저 스크립트에서 SDK for JavaScript를 사용하면 브라우저 설정이 사용되는 TLS 버전을 제어합니다. 브라우저에서 사용하는 TLS 버전은 스크립트로 검색하거나 설정할 수 없으며 사용자가 구성해야 합니다. 브라우저 스크립트에 사용된 TLS 버전을 확인하고 적용하려면 해당 브라우저의 지침을 참조하세요.

Microsoft Internet Explorer

1. Internet Explorer를 엽니다.
2. 메뉴 모음에서 도구 – 인터넷 옵션 – 고급 탭을 선택합니다.
3. 보안 범주까지 아래로 스크롤하여 TLS 1.2 사용 옵션 상자를 수동으로 선택합니다.
4. 확인을 클릭합니다.
5. 브라우저를 닫고 Internet Explorer를 다시 시작합니다.

Microsoft Edge

1. Windows 메뉴 검색 상자에 *### ##*을 입력합니다.
2. 가장 일치하는 항목에서 인터넷 옵션을 클릭합니다.
3. 인터넷 속성 창의 고급 탭에서 보안 섹션까지 아래로 스크롤합니다.
4. 사용자 TLS 1.2 확인란을 선택합니다.
5. 확인을 클릭합니다.

Google Chrome

1. Google Chrome을 엽니다.
2. Alt F를 클릭하고 설정을 선택합니다.
3. 아래로 스크롤하여 고급 설정 표시를 선택합니다.
4. 시스템 섹션까지 아래로 스크롤하여 프록시 설정 열기를 클릭합니다.
5. 고급 탭을 선택합니다.

6. 보안 범주까지 아래로 스크롤하여 TLS 1.2 사용 옵션 상자를 수동으로 선택합니다.
7. 확인을 클릭합니다.
8. 브라우저를 닫고 Google Chrome을 다시 시작합니다.

Mozilla Firefox

1. Firefox를 엽니다.
2. 주소 표시줄에 about:config를 입력하고 Enter 키를 누릅니다.
3. 검색 필드에 tls를 입력합니다. security.tls.version.min의 항목을 찾아 두 번 클릭합니다.
4. 정수 값을 3으로 설정하여 TLS 1.2의 프로토콜을 기본값으로 강제 설정합니다.
5. 확인을 클릭합니다.
6. 브라우저를 닫고 Mozilla Firefox를 다시 시작합니다.

Apple Safari

SSL 프로토콜을 활성화할 수 있는 옵션은 없습니다. Safari 버전 7 이상을 사용하는 경우 TLS 1.2가 자동으로 활성화됩니다.

추가 리소스

다음 링크는 AWS SDK for JavaScript에서 사용할 수 있는 추가 리소스를 제공합니다.

AWS SDK 및 도구 참조 가이드

AWS SDK 및 도구 참조 안내서에는 많은 AWS SDK에 공통적인 설정, 기능 및 기타 기본 개념도 포함되어 있습니다.

JavaScript SDK 포럼

SDK for JavaScript 포럼에서 SDK for JavaScript 사용자의 관심사에 대한 질문 및 토론을 찾아 볼 수 있습니다.

GitHub의 JavaScript SDK 및 개발자 안내서

GitHub에는 SDK for JavaScript에 사용할 수 있는 여러 리포지토리가 있습니다.

- 최신 SDK for JavaScript는 SDK 리포지토리에서 구할 수 있습니다.
- SDK for JavaScript 개발자 안내서(본 문서)는 자체 문서 리포지토리에서 마크다운 형식으로 제공됩니다.
- 이 가이드에 포함된 일부 샘플 코드는 SDK 샘플 코드 리포지토리에서 사용할 수 있습니다.

Gitter의 JavaScript SDK

Gitter의 SDK for JavaScript 커뮤니티에서도 SDK for JavaScript에 대한 질문과 토론을 찾을 수 있습니다.

AWS SDK for JavaScript 문서 기록

- SDK 버전: JavaScript API 참조 참조
- 주요 문서 업데이트 마지막 날짜: 2022년 3월 31일

문서 기록

다음 표에서는 2018년 5월 이후 AWS SDK for JavaScript의 각 릴리스에서 변경된 중요 사항에 대해 설명합니다. 이 설명서에 대한 업데이트 알림을 받으려면 RSS feed를 구독하세요.

변경 사항	설명	날짜
이제 모든 리전의 모든 AWS 서비스 API 엔드포인트에서 TLS 1.3 지원	지원되는 TLS 버전과 TLS 버전 로깅 방법이 업데이트되었습니다.	2025년 4월 10일
TLS의 최소 버전 적용	TLS 1.3에 관한 정보가 추가되었습니다.	2022년 3월 31일
브라우저에서 Amazon S3 버킷 내 사진 보기	기존 사진 앨범에서 사진을 보기만 하는 예제가 추가되었습니다.	2019년 5월 13일
새로운 자격 증명 로딩 방법인 Node.js에서 자격 증명 설정	ECS 자격 증명 공급자 또는 구성된 자격 증명 프로세스에서 로딩되는 자격 증명에 대한 정보가 추가되었습니다.	2019년 4월 25일
구성된 자격 증명 프로세스를 사용하는 자격 증명	구성된 자격 증명 프로세스에서 로딩되는 자격 증명에 대한 정보가 추가되었습니다.	2019년 4월 25일
새로운 브라우저 스크립트 시작	브라우저 스크립트에서 시작하는 방법이 다시 작성되어 예제를 간소화했고, 텍스트를 보내고 브라우저에서 재생할 수 있는 합성된 음성을 반환할 수 있	2018년 7월 14일

	도록 Amazon Polly 서비스에 액세스할 수 있게 되었습니다. 새로운 내용은 브라우저 스크립트에서 시작하기를 참조하세요.	
새 Amazon SNS 코드 샘플	Amazon SNS로 작업하기 위한 새 Node.js 코드 샘플 4개가 추가되었습니다. 예제 코드는 Amazon SNS 예제를 참조하세요.	2018년 6월 29일
새로운 Node.js 시작	Node.js에서 시작하기가 다시 작성되어 업데이트된 샘플 코드를 사용하고, `package.json` 파일 및 Node.js 코드를 생성하는 방법을 더 자세히 설명합니다. 새로운 내용은 Node.js에서 시작하기를 참조하세요.	2018년 6월 4일

이전 업데이트

다음 표에서는 2018년 6월 이전 AWS SDK for JavaScript의 각 릴리스에서 변경된 중요 사항에 대해 설명합니다.

변경 사항	설명	날짜
새 AWS Elemental MediaConvert 코드 샘플	AWS Elemental MediaConvert에 적용 가능한 새 Node.js 코드 샘플 3개가 추가되었습니다. 샘플 코드는 AWS Elemental MediaConvert 예제 섹션을 참조하세요.	2018년 5월 21일
GitHub의 새 편집 버튼	이제 모든 주제의 헤더에는 GitHub에서 동일한 주제의 마	2018년 2월 21일

변경 사항	설명	날짜
	크다운 버전으로 이동하는 버튼이 있어, 안내서의 정확성 및 완전성을 개선할 수 있는 편집 기능을 제공할 수 있습니다.	
사용자 지정 엔드포인트에 대한 새 주제	API 호출을 위한 사용자 지정 엔드포인트의 형식 및 사용에 대한 정보가 추가되었습니다. 사용자 지정 엔드포인트 지정을(를) 참조하세요.	2018년 2월 20일
GitHub의 SDK for JavaScript 개발자 안내서	SDK for JavaScript 개발자 안내서는 자체 문서 리포지토리에서 마크다운 형식으로 제공됩니다. 안내서에서 다루길 원하는 문제를 게시하거나 제안된 변경 사항을 제출하기 위해 풀 요청을 제출할 수 있습니다.	2018년 2월 16일
신규 Amazon DynamoDB 코드 샘플	문서 클라이언트를 사용한 DynamoDB 테이블 업데이트에 대한 새 Node.js 코드 샘플이 추가되었습니다. 샘플 코드는 DynamoDB 문서 클라이언트 사용 섹션을 참조하세요.	2018년 2월 14일
SDK 로깅에 대한 새로운 주제	타사 로거 사용에 대한 정보와 함께 SDK for JavaScript를 사용한 API 호출을 로깅하는 방법을 설명하는 주제가 추가되었습니다. AWS SDK for JavaScript 호출 로깅을(를) 참조하세요.	2018년 2월 5일

변경 사항	설명	날짜
리전 설정 주제가 업데이트됨	리전 설정 시 우선 순위에 관한 정보를 비롯해 SDK에서 사용되는 리전을 설정하는 방법을 설명하는 주제가 업데이트 및 확장되었습니다. AWS 리전 설정을(를) 참조하세요.	2017년 12월 12일
신규 Amazon SES 예제 코드	SDK 예제 코드가 포함된 섹션이 업데이트되어 Amazon SES 에서 적용할 수 있는 5가지 새 예제가 포함되었습니다. 이러한 코드 예제에 대한 자세한 내용은 Amazon Simple Email Services 예제 섹션을 참조하세요.	2017년 11월 9일

변경 사항	설명	날짜
사용성 개선	최근의 사용성 테스트를 바탕으로 설명서 사용성을 개선하기 위해 여러 가지가 변경되었습니다. • 브라우저 또는 Node.js 실행을 위한 대상으로 코드 샘플을 보다 명확하게 식별할 수 있습니다. • TOC 링크가 API 참조를 포함한 다른 웹 콘텐츠로 더 이상 직접 이동하지 않습니다. • 시작하기 섹션에서 AWS 자격 증명 획득에 대한 링크가 더 추가되었습니다. • SDK 사용에 필요한 일반적인 Node.js 기능에 대한 자세한 내용을 제공합니다. 자세한 내용은 Node.js 고려 사항 섹션을 참조하세요.	2017년 8월 9일
신규 DynamoDB 예제 코드	SDK 코드 예제가 포함된 섹션이 업데이트되어 이전 예제 2개를 다시 작성했고, DynamoDB에 적용할 수 있는 완전히 새로운 예제 3개가 추가되었습니다. 이러한 코드 예제에 대한 자세한 내용은 Amazon DynamoDB 예제 섹션을 참조하세요.	2017년 6월 21일

변경 사항	설명	날짜
신규 IAM 예제 코드	SDK 코드 예제가 포함된 섹션이 업데이트되어 IAM에 적용할 수 있는 5가지 새 예제가 포함되었습니다. 이러한 코드 예제에 대한 자세한 내용은 AWS IAM 예제 섹션을 참조하세요.	2016년 23월 12일
신규 CloudWatch 및 Amazon SQS 예제 코드	SDK 코드 예제가 포함된 섹션이 업데이트되어 CloudWatch 및 Amazon SQS에 적용할 수 있는 새 예제가 포함되었습니다. 이러한 코드 예제에 대한 자세한 내용은 Amazon CloudWatch 예제 및 Amazon SQS 예제 섹션을 참조하세요.	2016년 12월 20일
신규 Amazon EC2 예제 코드	SDK 코드 예제가 포함된 섹션이 업데이트되어 Amazon EC2에 적용할 수 있는 5가지 새 예제가 포함되었습니다. 이러한 코드 예제에 대한 자세한 내용은 Amazon EC2 예제 섹션을 참조하세요.	2016년 12월 15일
지원되는 브라우저 목록이 더욱 눈에 잘 띔	SDK for JavaScript에서 지원되는 브라우저 목록이 예전에는 사전 조건에 대한 주제에서 찾을 수 있었는데 별도 주제로 제공되어 목차에서 훨씬 쉽게 찾을 수 있게 되었습니다.	2016년 11월 16일

변경 사항	설명	날짜
최초로 발행된 새 개발자 안내서	이전 개발자 안내서는 이제 사용되지 않습니다. 새 개발자 안내서는 정보를 보다 쉽게 찾을 수 있도록 재구성되었습니다. Node.js 또는 브라우저 JavaScript 시나리오에서 특수한 고려 사항을 제시하는 경우 해당 고려 사항이 적절하게 식별됩니다. 또한 이 가이드는 빠르고 쉽게 찾을 수 있도록 추가 코드 예제를 더 나은 구성으로 제공합니다.	2016년 10월 28일