



Developer Guide for SDK Version 3

AWS SDK for JavaScript



AWS SDK for JavaScript: Developer Guide for SDK Version 3

Copyright © 2024 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

.....	xi
What's the AWS SDK for JavaScript?	1
Get started with the SDK	1
Maintenance and support for SDK major versions	2
Using the SDK with Node.js	2
Using the SDK with AWS Cloud9	2
Using the SDK with AWS Amplify	2
Using the SDK with web browsers	3
Using browsers in V3	3
Common use cases	3
About the examples	4
Resources	4
Get started	5
SDK authentication with AWS	5
Start an AWS access portal session	6
More authentication information	7
Get started with Node.js	8
The scenario	8
Prerequisites	8
Step 1: Set up the package structure and installing client packages	8
Step 2: Add necessary imports and SDK code	9
Step 3: Run the example	11
Get started in the browser	12
The Scenario	12
Step 1: Create an Amazon Cognito identity pool and IAM role	13
Step 2: Add a policy to the created IAM role	14
Step 3: Add a Amazon S3 bucket and object	14
Step 4: Set up the browser code	15
Step 5: Run the Example	16
Cleanup	17
Set up the SDK for JavaScript	18
Prerequisites	18
Set up an AWS Node.js environment	18
Supported web browsers	19

Install the SDK	20
Load the SDK	21
Configure the SDK for JavaScript	22
Configuration per service	22
Set configuration per service	23
Set the AWS Region	23
In a client class constructor	24
Use an environment variable	24
Use a shared config file	24
Order of precedence for setting the Region	24
Set credentials	25
Best practices for credentials	25
Set credentials in Node.js	26
Set credentials in a web browser	29
Node.js considerations	33
Use built-in Node.js modules	33
Use npm packages	33
Configure maxSockets in Node.js	34
Reuse connections with keep-alive in Node.js	35
Configure proxies for Node.js	35
Register certificate bundles in Node.js	36
Browser Script Considerations	37
Build the SDK for Browsers	37
Cross-origin resource sharing (CORS)	38
Bundle with webpack	42
Work with AWS services	46
Create and call service objects	46
Specify service object parameters	47
Call services asynchronously	47
Manage asynchronous calls	48
Use async/await	49
Use promises	50
Use a callback function	51
Create service client requests	52
Handle service client responses	53
Access data returned in the response	53

Access error information	54
Work with JSON	54
JSON as service object parameters	55
Code examples subset with guidance	56
JavaScript ES6/CommonJS syntax	57
Amazon DynamoDB examples	60
AWS Elemental MediaConvert examples	84
AWS Lambda examples	106
Amazon Lex examples	106
Amazon Polly examples	107
Amazon Redshift examples	110
Amazon SES examples	118
Amazon SNS Examples	145
Amazon Transcribe examples	180
Cross-service: Setting up Node.js on an Amazon EC2 instance	191
Cross-service: App to submit data	193
Cross-service: Amazon API Gateway and Lambda	201
Cross-service: Scheduled Lambda events	216
Cross-service: Amazon Lex example	228
Cross-service: Messaging app	242
Use AWS Cloud9 with the SDK for JavaScript	255
Step 1: Set up your AWS account to use AWS Cloud9	255
Step 2: Set up your AWS Cloud9 development environment	255
Step 3: Set up the SDK for JavaScript	256
To set up the SDK for JavaScript for Node.js	256
To set up the SDK for JavaScript in the browser	257
Step 4: Download example code	257
Step 5: Run and debug example code	257
Code examples	259
API Gateway	261
Scenarios	261
Aurora	262
Scenarios	261
Auto Scaling	263
Actions	264
Scenarios	261

Amazon Bedrock	306
Actions	264
Amazon Bedrock Runtime	311
Scenarios	261
AI21 Labs Jurassic-2	316
Amazon Titan Text	319
Anthropic Claude	325
Cohere Command	335
Meta Llama	338
Mistral AI	348
Agents for Amazon Bedrock	353
Actions	264
Agents for Amazon Bedrock Runtime	367
Actions	264
CloudWatch	369
Actions	264
CloudWatch Events	385
Actions	264
CloudWatch Logs	392
Actions	264
Scenarios	261
CodeBuild	410
Actions	264
Amazon Cognito Identity Provider	413
Actions	264
Scenarios	261
Amazon Comprehend	432
Scenarios	261
Amazon DocumentDB	438
Serverless examples	438
DynamoDB	439
Basics	441
Actions	264
Scenarios	261
Serverless examples	438
Amazon EC2	502

Basics	441
Actions	264
Scenarios	261
Elastic Load Balancing - Version 2	598
Actions	264
Scenarios	261
EventBridge	647
Actions	264
Scenarios	261
AWS Glue	655
Basics	441
Actions	264
HealthImaging	681
Actions	264
Scenarios	261
IAM	742
Actions	264
Scenarios	261
Kinesis	852
Serverless examples	438
Lambda	856
Actions	264
Scenarios	261
Serverless examples	438
Amazon Lex	892
Scenarios	261
Amazon MSK	893
Serverless examples	438
Amazon Personalize	894
Actions	264
Amazon Personalize Events	910
Actions	264
Amazon Personalize Runtime	914
Actions	264
Amazon Pinpoint	918
Actions	264

Amazon Polly	928
Scenarios	261
Amazon RDS	933
Scenarios	261
Serverless examples	438
Amazon RDS Data Service	937
Scenarios	261
Amazon Redshift	938
Actions	264
Amazon Rekognition	944
Scenarios	261
Amazon S3	947
Basics	441
Actions	264
Scenarios	261
Serverless examples	438
S3 Glacier	1021
Actions	264
SageMaker	1025
Actions	264
Scenarios	261
Secrets Manager	1063
Actions	264
Amazon SES	1065
Actions	264
Scenarios	261
Amazon SNS	1092
Actions	264
Scenarios	261
Serverless examples	438
Amazon SQS	1132
Actions	264
Scenarios	261
Serverless examples	438
Step Functions	1172
Actions	264

AWS STS	1173
Actions	264
AWS Support	1176
Basics	441
Actions	264
Amazon Textract	1194
Scenarios	261
Amazon Transcribe	1199
Actions	264
Scenarios	261
Amazon Translate	1208
Scenarios	261
Security	1215
Data protection	1215
Identity and Access Management	1216
Audience	1217
Authenticating with identities	1217
Managing access using policies	1221
How AWS services work with IAM	1223
Troubleshooting AWS identity and access	1223
Compliance Validation	1225
Resilience	1226
Infrastructure Security	1227
Enforce a minimum TLS version	1227
Verify and enforce TLS in Node.js	1228
Verify and enforce TLS in a browser script	1230
Migrate to v3	1233
Migrate to v3 using codemod	1233
Use codemod to migrate existing v2 code	1233
What's new in Version 3	1234
Modularized packages	1235
Comparing code size	1236
Calling commands in v3	1237
New middleware stack	1239
What's different between the v2 and v3	1240
Client constructors	1240

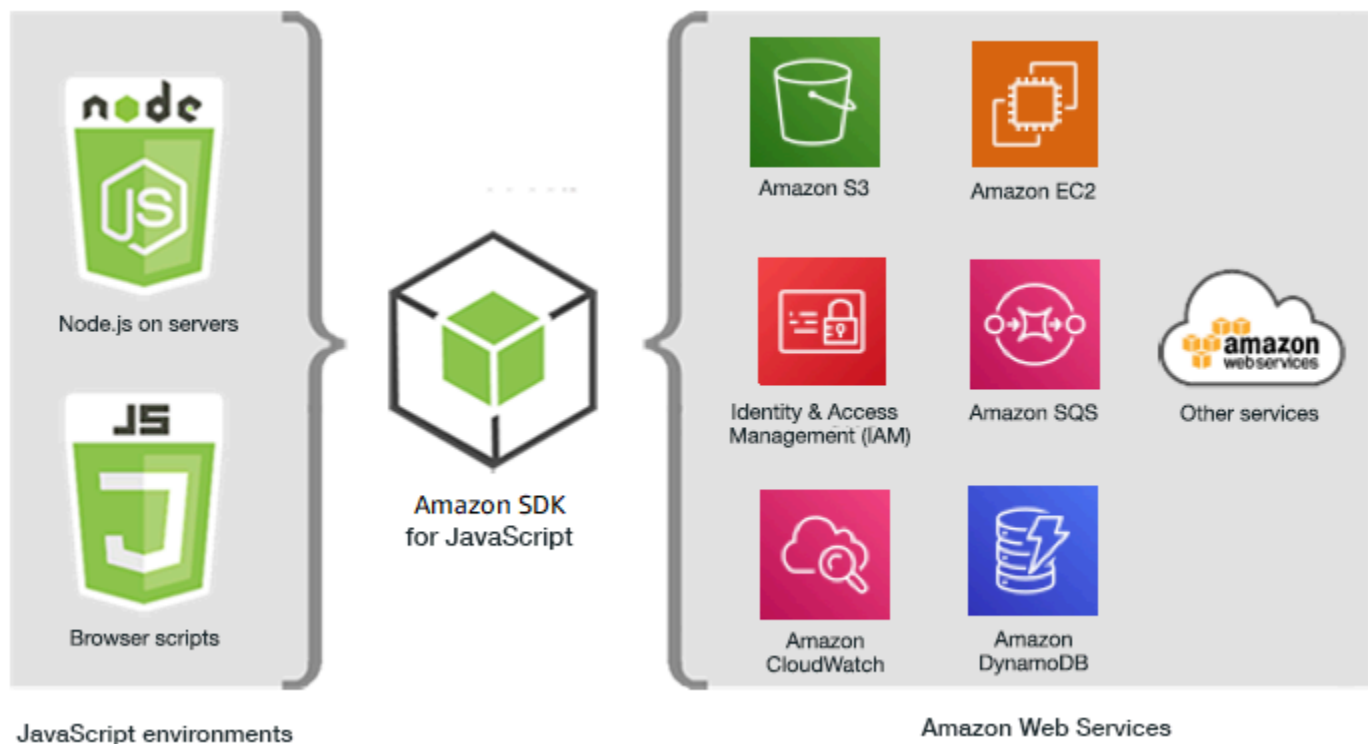
Credential providers	1245
Amazon S3 considerations	1251
DynamoDB document client	1253
Waiters and signers	1254
Notes on specific service clients	1256
Document history	1260
Document History	1260

The [AWS SDK for JavaScript V3 API Reference Guide](#) describes in detail all the API operations for the AWS SDK for JavaScript version 3 (V3).

What's the AWS SDK for JavaScript?

Welcome to the AWS SDK for JavaScript Developer Guide. This guide provides general information about setting up and configuring the AWS SDK for JavaScript. It also walks you through examples and tutorial of running various AWS services using the AWS SDK for JavaScript.

The [AWS SDK for JavaScript v3 API Reference Guide](#) provides a JavaScript API for AWS services. You can use the JavaScript API to build libraries or applications for [Node.js](#) or the browser.



Get started with the SDK

If you're ready to get hands-on with the SDK, follow the examples at [Get started](#).

To set up your development environment, see [Set up the SDK for JavaScript](#).

If you're currently using version 2.x of SDK for JavaScript, see [Migrate to v3](#) for specific guidance.

If you're looking for code examples for AWS services, see [SDK for JavaScript \(v3\) code examples](#).

Maintenance and support for SDK major versions

For information about maintenance and support for SDK major versions and their underlying dependencies, see the following in the [AWS SDKs and Tools Reference Guide](#):

- [AWS SDKs and tools maintenance policy](#)
- [AWS SDKs and tools version support matrix](#)

Using the SDK with Node.js

Node.js is a cross-platform runtime for running server-side JavaScript applications. You can set up Node.js on an Amazon Elastic Compute Cloud (Amazon EC2) instance to run on a server. You can also use Node.js to write on-demand AWS Lambda functions.

Using the SDK for Node.js differs from the way in which you use it for JavaScript in a web browser. The difference comes from the way in which you load the SDK and in how you obtain the credentials needed to access specific web services. When use of particular APIs differs between Node.js and the browser, we call out those differences.

Using the SDK with AWS Cloud9

You can also develop Node.js applications using the SDK for JavaScript in the AWS Cloud9 IDE. For more information about using AWS Cloud9 with the SDK for JavaScript, see [Use AWS Cloud9 with the AWS SDK for JavaScript](#).

Using the SDK with AWS Amplify

For browser-based web, mobile, and hybrid apps, you can also use the [AWS Amplify library on GitHub](#). It extends the SDK for JavaScript, providing a declarative interface.

Note

Frameworks such as Amplify might not offer the same browser support as the SDK for JavaScript. See the framework's documentation for details.

Using the SDK with web browsers

All major web browsers support execution of JavaScript. JavaScript code that is running in a web browser is often called *client-side JavaScript*.

For a list of browsers that are supported by the AWS SDK for JavaScript, see [Supported web browsers](#).

Using the SDK for JavaScript in a web browser differs from the way in which you use it for Node.js. The difference comes from the way in which you load the SDK and in how you obtain the credentials needed to access specific web services. When use of particular APIs differs between Node.js and the browser, we call out those differences.

Using browsers in V3

V3 enables you to bundle and include in the browser only the SDK for JavaScript files you require, reducing overhead.

To use V3 of the SDK for JavaScript in your HTML pages, you must bundle the required client modules and all required JavaScript functions into a single JavaScript file using Webpack, and add it in a script tag in the <head> of your HTML pages. For example:

```
<script src="./main.js"></script>
```

Note

For more information about Webpack, see [Bundle applications with webpack](#).

To use V2 of the SDK for JavaScript, you add a script tag that points to the latest version of the V2 SDK instead. For more information, see the [sample](#) in the AWS SDK for JavaScript Developer Guide v2.

Common use cases

Using the SDK for JavaScript in browser scripts makes it possible to realize a number of compelling use cases. Here are several ideas for things you can build in a browser application by using the SDK for JavaScript to access various web services.

- Build a custom console to AWS services in which you access and combine features across Regions and services to best meet your organizational or project needs.
- Use Amazon Cognito Identity to enable authenticated user access to your browser applications and websites, including use of third-party authentication from Facebook and others.
- Use Amazon Kinesis to process click streams or other marketing data in real time.
- Use Amazon DynamoDB for serverless data persistence, such as individual user preferences for website visitors or application users.
- Use AWS Lambda to encapsulate proprietary logic that you can invoke from browser scripts without downloading and revealing your intellectual property to users.

About the examples

You can browse the SDK for JavaScript examples in the [AWS Code Example Repository](#).

Resources

In addition to this guide, the following online resources are available for SDK for JavaScript developers:

- [AWS SDK for JavaScript V3 API Reference Guide](#)
- [AWS SDKs and Tools Reference Guide](#): Contains settings, features, and other foundational concepts common among AWS SDKs.
- [JavaScript Developer Blog](#)
- [AWS JavaScript Forum](#)
- [JavaScript examples in the AWS Code Catalog](#)
- [AWS Code Example Repository](#)
- [Gitter channel](#)
- [Stack Overflow](#)
- [Stack Overflow questions tagged AWS -sdk-js](#)
- GitHub
 - [SDK Source](#)
 - [Documentation Source](#)

Get started with the AWS SDK for JavaScript

The AWS SDK for JavaScript provides access to web services in either a browser or Node.js environment. This section has getting started exercises that show you how to work with the SDK for JavaScript in each of these JavaScript environments.

Note

You can develop Node.js applications, and JavaScript for browser-based applications, using the SDK for JavaScript in the AWS Cloud9 IDE. For an example of how to use AWS Cloud9 for Node.js development, see [Use AWS Cloud9 with the AWS SDK for JavaScript](#).

Topics

- [SDK authentication with AWS](#)
- [Get started with Node.js](#)
- [Get started in the browser](#)

SDK authentication with AWS

You must establish how your code authenticates with AWS when developing with AWS services. You can configure programmatic access to AWS resources in different ways depending on the environment and the AWS access available to you.

To choose your method of authentication and configure it for the SDK, see [Authentication and access](#) in the *AWS SDKs and Tools Reference Guide*.

We recommend that new users who are developing locally and are not given a method of authentication by their employer to set up AWS IAM Identity Center. This method includes installing the AWS CLI for ease of configuration and for regularly signing in to the AWS access portal. If you choose this method, your environment should contain the following elements after you complete the procedure for [IAM Identity Center authentication](#) in the *AWS SDKs and Tools Reference Guide*:

- The AWS CLI, which you use to start an AWS access portal session before you run your application.

- A [shared AWSconfig file](#) having a [default] profile with a set of configuration values that can be referenced from the SDK. To find the location of this file, see [Location of the shared files](#) in the *AWS SDKs and Tools Reference Guide*.
- The shared config file sets the [region](#) setting. This sets the default AWS Region that the SDK uses for AWS requests. This Region is used for SDK service requests that aren't specified with a Region to use.
- The SDK uses the profile's [SSO token provider configuration](#) to acquire credentials before sending requests to AWS. The `sso_role_name` value, which is an IAM role connected to an IAM Identity Center permission set, allows access to the AWS services used in your application.

The following sample config file shows a default profile set up with SSO token provider configuration. The profile's `sso_session` setting refers to the named [sso-session section](#). The `sso-session` section contains settings to initiate an AWS access portal session.

```
[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
```

The AWS SDK for JavaScript v3 does not need additional packages (such as SSO and SSO0IDC) to be added to your application to use IAM Identity Center authentication.

For details on using this credential provider explicitly, see [fromSSO\(\)](#) on the npm (Node.js package manager) website.

Start an AWS access portal session

Before running an application that accesses AWS services, you need an active AWS access portal session for the SDK to use IAM Identity Center authentication to resolve credentials. Depending on your configured session lengths, your access will eventually expire and the SDK will encounter an

authentication error. To sign in to the AWS access portal, run the following command in the AWS CLI.

```
aws sso login
```

If you followed the guidance and have a default profile setup, you do not need to call the command with a `--profile` option. If your SSO token provider configuration is using a named profile, the command is `aws sso login --profile named-profile`.

To optionally test if you already have an active session, run the following AWS CLI command.

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

If your session is active, the response to this command reports the IAM Identity Center account and permission set configured in the shared config file.

Note

If you already have an active AWS access portal session and run `aws sso login`, you will not be required to provide credentials.

The sign-in process might prompt you to allow the AWS CLI access to your data. Because the AWS CLI is built on top of the SDK for Python, permission messages might contain variations of the `botocore` name.

More authentication information

Human users, also known as *human identities*, are the people, administrators, developers, operators, and consumers of your applications. They must have an identity to access your AWS environments and applications. Human users that are members of your organization - that means you, the developer - are known as *workforce identities*.

Use temporary credentials when accessing AWS. You can use an identity provider for your human users to provide federated access to AWS accounts by assuming roles, which provide temporary credentials. For centralized access management, we recommend that you use AWS IAM Identity Center (IAM Identity Center) to manage access to your accounts and permissions within those accounts. For more alternatives, see the following:

- To learn more about best practices, see [Security best practices in IAM](#) in the *IAM User Guide*.

- To create short-term AWS credentials, see [Temporary Security Credentials](#) in the *IAM User Guide*.
- To learn about other AWS SDK for JavaScript V3 credential providers, see [Standardized credential providers](#) in the *AWS SDKs and Tools Reference Guide*.

Get started with Node.js

This guide shows you how to initialize an NPM package, add a service client to your package, and use the JavaScript SDK to call a service action.

The scenario

Create a new NPM package with one main file that does the following:

- Creates an Amazon Simple Storage Service bucket
- Puts an object in the Amazon S3 bucket
- Reads the object in the Amazon S3 bucket
- Confirms if the user wants to delete resources

Prerequisites

Before you can run the example, you must do the following:

- Configure your SDK authentication. For more information, see [SDK authentication with AWS](#).
- Install [Node.js](#).

Step 1: Set up the package structure and installing client packages

To set up the package structure and install the client packages:

1. Create a new folder `nodegetstarted` to contain the package.
2. From the command line, navigate to the new folder.
3. Run the following command to create a default `package.json` file:

```
npm init -y
```

4. Run the following command to install the Amazon S3 client package:

```
npm i @aws-sdk/client-s3
```

5. Add "type": "module" to the package.json file. This tells Node.js to use modern ESM syntax. The final package.json should look similar to the following:

```
{
  "name": "example-javascriptv3-get-started-node",
  "version": "1.0.0",
  "description": "This guide shows you how to initialize an NPM package, add a
  service client to your package, and use the JavaScript SDK to call a service
  action.",
  "main": "index.js",
  "scripts": {
    "test": "vitest run **/*.unit.test.js"
  },
  "author": "Your Name",
  "license": "Apache-2.0",
  "dependencies": {
    "@aws-sdk/client-s3": "^3.420.0"
  },
  "type": "module"
}
```

Step 2: Add necessary imports and SDK code

Add the following code to a file named `index.js` in the `nodegetstarted` folder.

```
// This is used for getting user input.
import { createInterface } from "readline/promises";

import {
  S3Client,
  PutObjectCommand,
  CreateBucketCommand,
  DeleteObjectCommand,
  DeleteBucketCommand,
  paginateListObjectsV2,
  GetObjectCommand,
} from "@aws-sdk/client-s3";
```

```
export async function main() {
  // A region and credentials can be declared explicitly. For example
  // `new S3Client({ region: 'us-east-1', credentials: {...} })` would
  // initialize the client with those settings. However, the SDK will
  // use your local configuration and credentials if those properties
  // are not defined here.
  const s3Client = new S3Client({});

  // Create an Amazon S3 bucket. The epoch timestamp is appended
  // to the name to make it unique.
  const bucketName = `test-bucket-${Date.now()}`;
  await s3Client.send(
    new CreateBucketCommand({
      Bucket: bucketName,
    })
  );

  // Put an object into an Amazon S3 bucket.
  await s3Client.send(
    new PutObjectCommand({
      Bucket: bucketName,
      Key: "my-first-object.txt",
      Body: "Hello JavaScript SDK!",
    })
  );

  // Read the object.
  const { Body } = await s3Client.send(
    new GetObjectCommand({
      Bucket: bucketName,
      Key: "my-first-object.txt",
    })
  );

  console.log(await Body.transformToString());

  // Confirm resource deletion.
  const prompt = createInterface({
    input: process.stdin,
    output: process.stdout,
  });

  const result = await prompt.question("Empty and delete bucket? (y/n) ");
}
```

```
prompt.close();

if (result === "y") {
  // Create an async iterator over lists of objects in a bucket.
  const paginator = paginateListObjectsV2(
    { client: s3Client },
    { Bucket: bucketName }
  );
  for await (const page of paginator) {
    const objects = page.Contents;
    if (objects) {
      // For every object in each page, delete it.
      for (const object of objects) {
        await s3Client.send(
          new DeleteObjectCommand({ Bucket: bucketName, Key: object.Key })
        );
      }
    }
  }

  // Once all the objects are gone, the bucket can be deleted.
  await s3Client.send(new DeleteBucketCommand({ Bucket: bucketName }));
}

// Call a function if this file was run directly. This allows the file
// to be runnable without running on import.
import { fileURLToPath } from "url";
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  main();
}
```

The example code can be found [here on GitHub](#).

Step 3: Run the example

Note

Remember to sign in! If you are using IAM Identity Center to authenticate, remember to sign in using the AWS CLI `aws sso login` command.

1. Run `node index.js`.
2. Choose whether to empty and delete the bucket.
3. If you don't delete the bucket, be sure to manually empty and delete it later.

Get started in the browser

This section walks you through an example that demonstrates how to run version 3 (V3) of the SDK for JavaScript in the browser.

Note

Running V3 in the browser is slightly different from version 2 (V2). For more information, see [Using browsers in V3](#).

For other examples of using (V3) of the SDK for JavaScript, see [SDK for JavaScript \(v3\) code examples](#).

This web application example shows you:

- How to access AWS services using Amazon Cognito for authentication.
- How to read a listing of objects in a Amazon Simple Storage Service (Amazon S3) bucket using a AWS Identity and Access Management (IAM) role.

Note

This example does not use AWS IAM Identity Center for authentication.

The Scenario

Amazon S3 is an object storage service that offers industry-leading scalability, data availability, security, and performance. You can use Amazon S3 to store data as objects within containers called buckets. For more information about Amazon S3, see the [Amazon S3 User Guide](#).

This example shows you how to set up and run a web app that assumes a IAM role to read from a Amazon S3 bucket. The example uses React front-end library and Vite front-end tooling to provide

a JavaScript development environment. The web app uses an Amazon Cognito identity pool to provide credentials needed to access AWS services. The included code example demonstrates the basic patterns for loading and using the SDK for JavaScript in web apps.

Step 1: Create an Amazon Cognito identity pool and IAM role

In this exercise, you create and use an Amazon Cognito identity pool to provide unauthenticated access to your web app for the Amazon S3 service. Creating an identity pool also creates a AWS Identity and Access Management (IAM) role to support unauthenticated guest users. For this example, we will only work with the unauthenticated user role to keep the task focused. You can integrate support for an identity provider and authenticated users later. For more information about adding a Amazon Cognito identity pool, see [Tutorial: Creating an identity pool](#) in the *Amazon Cognito Developer Guide*.

To create an Amazon Cognito identity pool and associated IAM role

1. Sign in to the AWS Management Console and open the Amazon Cognito console at <https://console.aws.amazon.com/cognito/>.
2. In the left navigation pane, choose **Identity pools**.
3. Choose **Create identity pool**.
4. In **Configure identity pool trust**, choose **Guest access** for user authentication.
5. In **Configure permissions**, choose **Create a new IAM role** and enter a name (for example, *getStartedRole*) in the **IAM role name**.
6. In **Configure properties**, enter a name (for example, *getStartedPool*) in **Identity pool name**.
7. In **Review and create**, confirm the selections that you made for your new identity pool. Select **Edit** to return to the wizard and change any settings. When you're done, select **Create identity pool**.
8. Note the **Identity pool ID** and the **Region** of the newly created Amazon Cognito identity pool. You need these values to replace *IDENTITY_POOL_ID* and *REGION* in [Step 4: Set up the browser code](#).

After you create your Amazon Cognito identity pool, you're ready to add permissions for Amazon S3 that are needed by your web app.

Step 2: Add a policy to the created IAM role

To enable access to a Amazon S3 bucket in your web app, use the unauthenticated IAM role (for example, *getStartedRole*) created for your Amazon Cognito identity pool (for example, *getStartedPool*). This requires you to attach an IAM policy to the role. For more information about modifying IAM roles, see [Modifying a role permissions policy](#) in the *IAM User Guide*.

To add an Amazon S3 policy to the IAM role associated with unauthenticated users

1. Sign in to the AWS Management Console and open the IAM console at <https://console.aws.amazon.com/iam/>.
2. In the left navigation pane, choose **Roles**.
3. Choose the name of the role that you want to modify (for example, *getStartedRole*), and then choose the **Permissions** tab.
4. Choose **Add permissions** and then choose **Attach policies**.
5. In the **Add permissions** page for this role, find and then select the check box for **AmazonS3ReadOnlyAccess**.

Note

You can use this process to enable access to any AWS service.

6. Choose **Add permissions**.

After you create your Amazon Cognito identity pool and add permissions for Amazon S3 to your IAM role for unauthenticated users, you are ready to add and configure a Amazon S3 bucket.

Step 3: Add a Amazon S3 bucket and object

In this step, you will add a Amazon S3 bucket and object for the example. You will also enable cross-origin resource sharing (CORS) for the bucket. For more information about creating Amazon S3 buckets and objects, see [Getting started with Amazon S3](#) in the *Amazon S3 User Guide*.

To add an Amazon S3 bucket and object with CORS

1. Sign in to the AWS Management Console and open the Amazon S3 console at <https://console.aws.amazon.com/s3/>.

2. In the left navigation pane, choose **Buckets** and choose **Create bucket**.
3. Enter a bucket name that conforms to [bucket naming rules](#) (for example, *getstartedbucket*) and choose **Create bucket**.
4. Choose the bucket you created, and then choose the **Objects** tab. Then choose **Upload**.
5. Under **Files and folders**, choose **Add files**.
6. Choose a file to upload, and then choose **Open**. Then choose **Upload** to complete uploading the object to your bucket.
7. Next, choose the **Permissions** tab of your bucket, and then choose **Edit** in the **Cross-origin resource sharing (CORS)** section. Enter the following JSON:

```
[
  {
    "AllowedHeaders": [
      "*"
    ],
    "AllowedMethods": [
      "GET"
    ],
    "AllowedOrigins": [
      "*"
    ],
    "ExposeHeaders": []
  }
]
```

8. Choose **Save changes**.

After you have added a Amazon S3 bucket and added an object, you're ready to set up the browser code.

Step 4: Set up the browser code

The example application consists of a single-page React application. The files for this example can be found [here on GitHub](#).

To set up the example application

1. Install [Node.js](#).
2. From the command line, clone the [AWS Code Examples Repository](#):

```
git clone --depth 1 https://github.com/awsdocs/aws-doc-sdk-examples.git
```

3. Navigate to the example application:

```
cd aws-doc-sdk-examples/javascriptv3/example_code/web/s3/list-objects/
```

4. Run the following command to install the required packages:

```
npm install
```

5. Next, open `src/App.tsx` in a text editor and complete the following:

- Replace `YOUR_IDENTITY_POOL_ID` with the Amazon Cognito identity pool ID you noted in [Step 1: Create an Amazon Cognito identity pool and IAM role](#).
- Replace the value for region to the region assigned for your Amazon S3 bucket and Amazon Cognito identity pool. Note that the regions for both service must be the same (for example, `us-east-2`).
- Replace `bucket-name` with bucket name you created in [Step 3: Add a Amazon S3 bucket and object](#).

After you have replaced the text, save the `App.tsx` file. You're now ready to run the web app.

Step 5: Run the Example

To run the example application

1. From the command line, navigate to the example application:

```
cd aws-doc-sdk-examples/javascriptv3/example_code/web/s3/list-objects/
```

2. From the command line, run the following command:

```
npm run dev
```

The Vite development environment will run with the following message:

```
VITE v4.3.9 ready in 280 ms
```

```
# Local: http://localhost:5173/  
# Network: use --host to expose  
# press h to show help
```

3. In your web browser, navigate to the URL shown above (for example, <http://localhost:5173/>). The example app will show you a list of object filenames in your Amazon S3 bucket.

Cleanup

To clean up the resources you created during this tutorial, do the following:

- In [the Amazon S3 console](#), delete any objects and any buckets created (for example, *getstartedbucket*).
- In [the IAM console](#), delete the role name (for example, *getStartedRole*).
- In [the Amazon Cognito console](#), delete the identity pool name (for example, *getStartedPool*).

Set up the SDK for JavaScript

The topics in this section explain how to install and load the SDK for JavaScript so you can access the web services supported by the SDK.

Note

React Native developers should use AWS Amplify to create new projects on AWS. See the [aws-sdk-react-native](#) archive for details.

Topics

- [Prerequisites](#)
- [Install the SDK for JavaScript](#)
- [Load the SDK for JavaScript](#)

Prerequisites

Install Node.js on your servers, if it's not already installed.

Topics

- [Set up an AWS Node.js environment](#)
- [Supported web browsers](#)

Set up an AWS Node.js environment

To set up an AWS Node.js environment in which you can run your application, use any of the following methods:

- Choose an Amazon Machine Image (AMI) with Node.js preinstalled. Then create an Amazon EC2 instance using that AMI. When creating your Amazon EC2 instance, choose your AMI from the AWS Marketplace. Search the AWS Marketplace for Node.js and choose an AMI option that includes a preinstalled version of Node.js (32-bit or 64-bit).
- Create an Amazon EC2 instance and install Node.js on it. For more information about how to install Node.js on an Amazon Linux instance, see [Setting up Node.js on an Amazon EC2 instance](#).

- Create a serverless environment using AWS Lambda to run Node.js as a Lambda function. For more information about using Node.js within a Lambda function, see [Programming model \(Node.js\)](#) in the *AWS Lambda Developer Guide*.
- Deploy your Node.js application to AWS Elastic Beanstalk. For more information about using Node.js with Elastic Beanstalk, see [Deploying Node.js applications to AWS Elastic Beanstalk](#) in the *AWS Elastic Beanstalk Developer Guide*.
- Create a Node.js application server using AWS OpsWorks. For more information about using Node.js with AWS OpsWorks, see [Creating your first Node.js stack](#) in the *AWS OpsWorks User Guide*.

Supported web browsers

The AWS SDK for JavaScript supports all modern web browsers.

In version 3.183.0 or later, the SDK for JavaScript uses ES2020 artifacts, which supports the following minimum versions.

Browser	Version
Google Chrome	80.0+
Mozilla Firefox	80.0+
Opera	63.0+
Microsoft Edge	80.0+
Apple Safari	14.1+
Samsung Internet	12.0+

In version 3.182.0 or earlier, the SDK for JavaScript uses ES5 artifacts, which supports the following minimum versions.

Browser	Version
Google Chrome	49.0+

Browser	Version
Mozilla Firefox	45.0+
Opera	36.0+
Microsoft Edge	12.0+
Windows Internet Explorer	N/A
Apple Safari	9.0+
Android Browser	76.0+
UC Browser	12.12+
Samsung Internet	5.0+

Note

Frameworks such as AWS Amplify might not offer the same browser support as the SDK for JavaScript. See the [AWS Amplify Documentation](#) for details.

Install the SDK for JavaScript

Not all services are immediately available in the SDK or in all AWS Regions.

To install a service from the AWS SDK for JavaScript using [npm, the Node.js package manager](#), enter the following command at the command prompt, where *SERVICE* is the name of a service, such as s3.

```
npm install @aws-sdk/client-SERVICE
```

For a full list of the AWS SDK for JavaScript service client packages, see the [AWS SDK for JavaScript API Reference guide](#).

Load the SDK for JavaScript

After you install the SDK, you can load a client package in your node application using `import`. For example, to load the Amazon S3 client and the Amazon S3 [ListBuckets](#) command, use the following.

```
import { S3Client, ListBucketsCommand } from "@aws-sdk/client-s3";
```


Configure the SDK for JavaScript

Before you use the SDK for JavaScript to invoke web services using the API, you must configure the SDK. At a minimum, you must configure:

- The AWS Region in which you will request services
- How your code authenticates with AWS

In addition to these settings, you might also have to configure permissions for your AWS resources. For example, you can limit access to an Amazon S3 bucket or restrict an Amazon DynamoDB table for read-only access.

The [AWS SDKs and Tools Reference Guide](#) also contains settings, features, and other foundational concepts common among many of the AWS SDKs.

The topics in this section describe the ways to configure the SDK for JavaScript for Node.js and JavaScript running in a web browser.

Topics

- [Configuration per service](#)
- [Set the AWS Region](#)
- [Set credentials](#)
- [Node.js considerations](#)
- [Browser Script Considerations](#)

Configuration per service

You can configure the SDK by passing configuration information to a service object.

Service-level configuration provides significant control over individual services, enabling you to update the configuration of individual service objects when your needs vary from the default configuration.

Note

In version 2.x of the AWS SDK for JavaScript service configuration could be passed to individual client constructors. However, these configurations would first be merged automatically into a copy of the global SDK configuration `AWS.config`. Also, calling `AWS.config.update({/* params */})` only updated configuration for service clients instantiated after the update call was made, not any existing clients. This behavior was a frequent source of confusion, and made it difficult to add configuration to the global object that only affects a subset of service clients in a forward-compatible way. In version 3, there is no longer a global configuration managed by the SDK. Configuration must be passed to each service client that is instantiated. It is still possible to share the same configuration across multiple clients but that configuration will not be automatically merged with a global state.

Set configuration per service

Each service that you use in the SDK for JavaScript is accessed through a service object that is part of the API for that service. For example, to access the Amazon S3 service, you create the Amazon S3 service object. You can specify configuration settings that are specific to a service as part of the constructor for that service object.

For example, if you need to access Amazon EC2 objects in multiple AWS Regions, create an Amazon EC2 service object for each Region and then set the Region configuration of each service object accordingly.

```
var ec2_regionA = new EC2({region: 'ap-southeast-2', maxAttempts: 15});
var ec2_regionB = new EC2({region: 'us-west-2', maxAttempts: 15});
```

Set the AWS Region

An AWS Region is a named set of AWS resources in the same geographical area. An example of a Region is `us-east-1`, which is the US East (N. Virginia) Region. You specify a Region when creating a service client in the SDK for JavaScript so that the SDK accesses the service in that Region. Some services are available only in specific Regions.

The SDK for JavaScript doesn't select a Region by default. However, you can set the AWS Region using an environment variable, or a shared configuration `config` file.

In a client class constructor

When you instantiate a service object, you can specify the AWS Region for that resource as part of the client class constructor, as shown here.

```
const s3Client = new S3.S3Client({region: 'us-west-2'});
```

Use an environment variable

You can set the Region using the `AWS_REGION` environment variable. If you define this variable, the SDK for JavaScript reads it and uses it.

Use a shared config file

Much like the shared credentials file lets you store credentials for use by the SDK, you can keep your AWS Region and other configuration settings in a shared file named `config` for the SDK to use. If the `AWS_SDK_LOAD_CONFIG` environment variable is set to a truthy value, the SDK for JavaScript automatically searches for a `config` file when it loads. Where you save the `config` file depends on your operating system:

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

If you don't already have a shared `config` file, you can create one in the designated directory. In the following example, the `config` file sets both the Region and the output format.

```
[default]
region=us-west-2
output=json
```

For more information about using shared `config` and `credentials` files, see [Shared config and credentials files](#) in the *AWS SDKs and Tools Reference Guide*.

Order of precedence for setting the Region

The following is the order of precedence for Region setting:

1. If a Region is passed to a client class constructor, that Region is used.

2. If a Region is set in the environment variable, that Region is used.
3. Otherwise, the Region defined in the shared config file is used.

Set credentials

AWS uses credentials to identify who is calling services and whether access to the requested resources is allowed.

Whether running in a web browser or in a Node.js server, your JavaScript code must obtain valid credentials before it can access services through the API. Credentials can be set per service, by passing credentials directly to a service object.

There are several ways to set credentials that differ between Node.js and JavaScript in web browsers. The topics in this section describe how to set credentials in Node.js or web browsers. In each case, the options are presented in recommended order.

Best practices for credentials

Properly setting credentials ensures that your application or browser script can access the services and resources needed while minimizing exposure to security issues that may impact mission critical applications or compromise sensitive data.

An important principle to apply when setting credentials is to always grant the least privilege required for your task. It's more secure to provide minimal permissions on your resources and add further permissions as needed, rather than provide permissions that exceed the least privilege and, as a result, be required to fix security issues you might discover later. For example, unless you have a need to read and write individual resources, such as objects in an Amazon S3 bucket or a DynamoDB table, set those permissions to read only.

For more information about granting the least privilege, see the [Grant least privilege](#) section of the Best Practices topic in the *IAM User Guide*.

Topics

- [Set credentials in Node.js](#)
- [Set credentials in a web browser](#)

Set credentials in Node.js

We recommend that new users who are developing locally and are not given a method of authentication by their employer to set up AWS IAM Identity Center. For more information, see [SDK authentication with AWS](#).

There are several ways in Node.js to supply your credentials to the SDK. Some of these are more secure and others afford greater convenience while developing an application. When obtaining credentials in Node.js, be careful about relying on more than one source, such as an environment variable and a JSON file you load. You can change the permissions under which your code runs without realizing the change has happened.

AWS SDK for JavaScript V3 provides a default credential provider chain in Node.js, so you are not required to supply a credential provider explicitly. The default [credential provider chain](#) attempts to resolve the credentials from a variety of different sources in a given precedence, until a credential is returned from the one of the sources. You can find the credential provider chain for SDK for JavaScript V3 [here](#).

Credential provider chain

All SDKs have a series of places (or sources) that they check in order to get valid credentials to use to make a request to an AWS service. After valid credentials are found, the search is stopped. This systematic search is called the default credential provider chain.

For each step in the chain, there are different ways to set the values. Setting values directly in code always takes precedence, followed by setting as environment variables, and then in the shared AWS config file. For more information, see [Precedence of settings](#) in the *AWS SDKs and Tools Reference Guide*.

The *AWS SDKs and Tools Reference Guide* has information on SDK configuration settings used by all AWS SDKs and the AWS CLI. To learn more about how to configure the SDK through the shared AWS config file, see [Shared config and credentials files](#). To learn more about how to configure the SDK through setting environment variables, see [Environment variables support](#).

To authenticate with AWS, the AWS SDK for JavaScript checks the credential providers in the order listed in the following table.

<i>AWS SDK for JavaScript API Reference</i> credential provider method by precedence	Credential provider(s) available	<i>AWS SDKs and Tools Reference Guide</i>
<code>fromEnv()</code>	AWS access keys from environment variables	AWS access keys
<code>fromSSO()</code>	AWS IAM Identity Center. In this guide, see SDK authentication with AWS .	IAM Identity Center credential provider
<code>fromIni()</code>	AWS access keys from shared config and credentials files	AWS access keys
	Trusted entity provider (such as <code>AWS_ROLE_ARN</code>)	Assume an IAM role
	Web identity token from AWS Security Token Service (AWS STS)	Federate with web identity or OpenID Connect
	Amazon Elastic Container Service (Amazon ECS) credentials	Container credential provider
	Amazon Elastic Compute Cloud (Amazon EC2) instance profile credentials (IMDS credential provider)	IMDS credential provider
	Process credential provider	Process credential provider
	AWS IAM Identity Center credentials	IAM Identity Center credential provider
<code>fromProcess()</code>	Process credential provider	Process credential provider

<i>AWS SDK for JavaScript API Reference</i> credential provider method by precedence	Credential provider(s) available	<i>AWS SDKs and Tools Reference Guide</i>
<u>fromTokenFile()</u>	Web identity token from AWS Security Token Service (AWS STS)	<u>Federate with web identity or OpenID Connect</u>
<u>fromContainerMetadata()</u>	Amazon Elastic Container Service (Amazon ECS) credentials	<u>Container credential provider</u>
<u>fromInstanceMetadata()</u>	Amazon Elastic Compute Cloud (Amazon EC2) instance profile credentials (IMDS credential provider)	<u>IMDS credential provider</u>

If you followed the recommended approach for new users to get started, you set up AWS IAM Identity Center authentication during [SDK authentication with AWS](#) of the Getting started topic. Other authentication methods are useful for different situations. To avoid security risks, we recommend always using short-term credentials. For other authentication method procedures, see [Authentication and access](#) in the *AWS SDKs and Tools Reference Guide*.

The topics in this section describe how to load credentials into Node.js.

Topics

- [Load credentials in Node.js from IAM roles for Amazon EC2](#)
- [Load credentials for a Node.js Lambda function](#)

Load credentials in Node.js from IAM roles for Amazon EC2

If you run your Node.js application on an Amazon EC2 instance, you can leverage IAM roles for Amazon EC2 to automatically provide credentials to the instance. If you configure your instance to use IAM roles, the SDK automatically selects the IAM credentials for your application, eliminating the need to manually provide credentials.

For more information about adding IAM roles to an Amazon EC2 instance, see [IAM roles for Amazon EC2](#).

Load credentials for a Node.js Lambda function

When you create an AWS Lambda function, you must create a special IAM role that has permission to execute the function. This role is called the *execution role*. When you set up a Lambda function, you must specify the IAM role you created as the corresponding execution role.

The execution role provides the Lambda function with the credentials it needs to run and to invoke other web services. As a result, you don't need to provide credentials to the Node.js code you write within a Lambda function.

For more information about creating a Lambda execution role, see [Manage permissions: Using an IAM role \(execution role\)](#) in the *AWS Lambda Developer Guide*.

Set credentials in a web browser

There are several ways to supply your credentials to the SDK from browser scripts. Some of these are more secure and others afford greater convenience while developing a script.

Here are the ways you can supply your credentials, in order of recommendation:

1. Using Amazon Cognito Identity to authenticate users and supply credentials
2. Using web federated identity

Warning

We do not recommend hard coding your AWS credentials in your scripts. Hard coding credentials poses a risk of exposing your access key ID and secret access key.

Topics

- [Use Amazon Cognito Identity to authenticate users](#)

Use Amazon Cognito Identity to authenticate users

The recommended way to obtain AWS credentials for your browser scripts is to use the Amazon Cognito Identity credentials client `CognitoIdentityClient`. Amazon Cognito enables authentication of users through third-party identity providers.

To use Amazon Cognito Identity, you must first create an identity pool in the Amazon Cognito console. An identity pool represents the group of identities that your application provides to your users. The identities given to users uniquely identify each user account. Amazon Cognito identities are not credentials. They are exchanged for credentials using web identity federation support in AWS Security Token Service (AWS STS).

Amazon Cognito helps you manage the abstraction of identities across multiple identity providers. The identity that is loaded is then exchanged for credentials in AWS STS.

Configure the Amazon Cognito Identity credentials object

If you have not yet created one, create an identity pool to use with your browser scripts in the [Amazon Cognito console](#) before you configure your Amazon Cognito client. Create and associate both authenticated and unauthenticated IAM roles for your identity pool. For more information, see [Tutorial: Creating an identity pool](#) in the *Amazon Cognito Developer Guide*.

Unauthenticated users don't have their identity verified, making this role appropriate for guest users of your app or in cases when it doesn't matter if users have their identities verified. Authenticated users log in to your application through a third-party identity provider that verifies their identities. Make sure you scope the permissions of resources appropriately so you don't grant access to them from unauthenticated users.

After you configure an identity pool, use the `fromCognitoIdentityPool` method from the `@aws-sdk/credential-providers` to retrieve the credentials from the identity pool. In the following example of creating an Amazon S3 client, replace `AWS_REGION` with the region and `IDENTITY_POOL_ID` with the identity pool ID.

```
// Import required AWS SDK clients and command for Node.js
import {S3Client} from "@aws-sdk/client-s3";
import {fromCognitoIdentityPool} from "@aws-sdk/credential-providers";

const REGION = AWS_REGION;
```

```
const s3Client = new S3Client({
  region: REGION,
  credentials: fromCognitoIdentityPool({
    clientConfig: { region: REGION }, // Configure the underlying
    CognitoIdentityClient.
    identityPoolId: 'IDENTITY_POOL_ID',
    logins: {
      // Optional tokens, used for authenticated login.
    },
  })
});
```

The optional `logins` property is a map of identity provider names to the identity tokens for those providers. How you get the token from your identity provider depends on the provider you use. For example, if you are using an Amazon Cognito user pool as your authentication provider, you could use a method similar to the one below.

```
// Get the Amazon Cognito ID token for the user. 'getToken()' below.
let idToken = getToken();
let COGNITO_ID = "COGNITO_ID"; // 'COGNITO_ID' has the format 'cognito-
idp.REGION.amazonaws.com/COGNITO_USER_POOL_ID'
let loginData = {
  [COGNITO_ID]: idToken,
};
const s3Client = new S3Client({
  region: REGION,
  credentials: fromCognitoIdentityPool({
    clientConfig: { region: REGION }, // Configure the underlying
    CognitoIdentityClient.
    identityPoolId: 'IDENTITY_POOL_ID',
    logins: loginData
  })
});

// Strips the token ID from the URL after authentication.
window.getToken = function () {
  var idtoken = window.location.href;
  var idtoken1 = idtoken.split("=")[1];
  var idtoken2 = idtoken1.split("&")[0];
  var idtoken3 = idtoken2.split("&")[0];
  return idtoken3;
};
```

Switch Unauthenticated Users to Authenticated Users

Amazon Cognito supports both authenticated and unauthenticated users. Unauthenticated users receive access to your resources even if they aren't logged in with any of your identity providers. This degree of access is useful to display content to users prior to logging in. Each unauthenticated user has a unique identity in Amazon Cognito even though they have not been individually logged in and authenticated.

Initially Unauthenticated User

Users typically start with the unauthenticated role, for which you set the credentials property of your configuration object without a logins property. In this case, your default credentials might look like the following:

```
// Import the required AWS SDK for JavaScript v3 modules.
import {fromCognitoIdentityPool} from "@aws-sdk/credential-providers";
// Set the default credentials.
const creds = fromCognitoIdentityPool({
  identityPoolId: 'IDENTITY_POOL_ID',
  clientConfig: { region: REGION } // Configure the underlying CognitoIdentityClient.
});
```

Switch to Authenticated User

When an unauthenticated user logs in to an identity provider and you have a token, you can switch the user from unauthenticated to authenticated by calling a custom function that updates the credentials object and adds the logins token.

```
// 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;
}
```

Node.js considerations

Although Node.js code is JavaScript, using the AWS SDK for JavaScript in Node.js can differ from using the SDK in browser scripts. Some API methods work in Node.js but not in browser scripts, as well as the other way around. And successfully using some APIs depends on your familiarity with common Node.js coding patterns, such as importing and using other Node.js modules like the File System (`fs`) module.

Use built-in Node.js modules

Node.js provides a collection of built-in modules you can use without installing them. To use these modules, create an object with the `require` method to specify the module name. For example, to include the built-in HTTP module, use the following.

```
import http from 'http';
```

Invoke methods of the module as if they are methods of that object. For example, here is code that reads an HTML file.

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

For a complete list of all built-in modules that Node.js provides, see [Node.js documentation](#) on the Node.js website.

Use npm packages

In addition to the built-in modules, you can also include and incorporate third-party code from npm, the Node.js package manager. This is a repository of open source Node.js packages and a command-line interface for installing those packages. For more information about npm and a list

of currently available packages, see <https://www.npmjs.com>. You can also learn about additional Node.js packages you can use [here on GitHub](#).

Configure maxSockets in Node.js

In Node.js, you can set the maximum number of connections per origin. If `maxSockets` is set, the low-level HTTP client queues requests and assigns them to sockets as they become available.

This lets you set an upper bound on the number of concurrent requests to a given origin at a time. Lowering this value can reduce the number of throttling or timeout errors received. However, it can also increase memory usage because requests are queued until a socket becomes available.

The following example shows how to set `maxSockets` for a DynamoDB client.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { NodeHttpHandler } from "@smithy/node-http-handler";
import https from "https";
let agent = new https.Agent({
  maxSockets: 25
});

let dynamodbClient = new DynamoDBClient({
  requestHandler: new NodeHttpHandler({
    requestTimeout: 3_000,
    httpsAgent: agent
  });
});
```

The SDK for JavaScript uses a `maxSockets` value of 50 if you do not supply a value or an Agent object. If you supply an Agent object, its `maxSockets` value will be used. For more information about setting `maxSockets` in Node.js, see the [Node.js documentation](#).

As of v3.521.0 of the AWS SDK for JavaScript, you can use the following [shorthand syntax](#) to configure `requestHandler`.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({
  requestHandler: {
    requestTimeout: 3_000,
    httpsAgent: { maxSockets: 25 },
  },
});
```

```
});
```

Reuse connections with keep-alive in Node.js

The default Node.js HTTP/HTTPS agent creates a new TCP connection for every new request. To avoid the cost of establishing a new connection, the AWS SDK for JavaScript reuses TCP connections *by default*.

For short-lived operations, such as Amazon DynamoDB queries, the latency overhead of setting up a TCP connection might be greater than the operation itself. Additionally, since DynamoDB [encryption at rest](#) is integrated with [AWS KMS](#), you may experience latencies from the database having to re-establish new AWS KMS cache entries for each operation.

If you do not want to reuse TCP connections, you can disable reusing these connections alive with `keepAlive` on a per-service client basis as shown in the following example for a DynamoDB client.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { NodeHttpHandler } from "@smithy/node-http-handler";
import { Agent } from "https";

const dynamodbClient = new DynamoDBClient({
  requestHandler: new NodeHttpHandler({
    httpsAgent: new Agent({ keepAlive: false })
  })
});
```

If `keepAlive` is enabled, you can also set the initial delay for TCP Keep-Alive packets with `keepAliveMsecs`, which by default is 1000 ms. See the [Node.js documentation](#) for details.

Configure proxies for Node.js

If you can't directly connect to the internet, the SDK for JavaScript supports use of HTTP or HTTPS proxies through a third-party HTTP agent.

To find a third-party HTTP agent, search for "HTTP proxy" at [npm](#).

To install a third-party HTTP agent proxy, enter the following at the command prompt, where **PROXY** is the name of the npm package.

```
npm install PROXY --save
```

To use a proxy in your application, use the `httpAgent` and `httpsAgent` property, as shown in the following example for a DynamoDB client.

```
import { DynamoDBClient } from '@aws-sdk/client-dynamodb';
import { NodeHttpHandler } from "@smithy/node-http-handler";
import { HttpsProxyAgent } from "hpagent";
const agent = new HttpsProxyAgent({ proxy: "http://internal.proxy.com" });
const dynamodbClient = new DynamoDBClient({
  requestHandler: new NodeHttpHandler({
    httpAgent: agent,
    httpsAgent: agent
  }),
});
```

Note

`httpAgent` is not the same as `httpsAgent`, and since most calls from the client will be to `https`, both should be set.

Register certificate bundles in Node.js

The default trust stores for Node.js include the certificates needed to access AWS services. In some cases, it might be preferable to include only a specific set of certificates.

In this example, a specific certificate on disk is used to create an `https.Agent` that rejects connections unless the designated certificate is provided. The newly created `https.Agent` is then used by the DynamoDB client.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { NodeHttpHandler } from "@smithy/node-http-handler";
import { Agent } from "https";
import { readFileSync } from "fs";
const certs = [readFileSync("/path/to/cert.pem")];
const agent = new Agent({
  rejectUnauthorized: true,
  ca: certs
});
const dynamodbClient = new DynamoDBClient({
  requestHandler: new NodeHttpHandler({
    httpAgent: agent,
```

```
    httpsAgent: agent
  })
});
```

Browser Script Considerations

The following topics describe special considerations for using the AWS SDK for JavaScript in browser scripts.

Topics

- [Build the SDK for Browsers](#)
- [Cross-origin resource sharing \(CORS\)](#)
- [Bundle applications with webpack](#)

Build the SDK for Browsers

Unlike SDK for JavaScript version 2 (V2), V3 is not provided as a JavaScript file with support included for a default set of services. Instead V3 enables you to bundle and include in the browser only the SDK for JavaScript files you require, reducing overhead. We recommend using Webpack to bundle the required SDK for JavaScript files, and any additional third-party packages you require, into a single JavaScript file, and load it into browser scripts using a `<script>` tag. For more information about Webpack, see [Bundle applications with webpack](#). For an example that uses Webpack to load V3 SDK for JavaScript into a browser, see [Build an app to submit data to DynamoDB](#).

If you work with the SDK outside of an environment that enforces CORS in your browser and if you want access to all services provided by the SDK for JavaScript, you can build a custom copy of the SDK locally by cloning the repository and running the same build tools that build the default hosted version of the SDK. The following sections describe the steps to build the SDK with extra services and API versions.

Use the SDK Builder to build the SDK for JavaScript

Note

Amazon Web Services version 3 (V3) no longer supports Browser Builder. To minimize bandwidth usage of browser applications, we recommend you import named modules, and

bundle them to reduce size. For more information about bundling, see [Bundle applications with webpack](#).

Cross-origin resource sharing (CORS)

Cross-origin resource sharing, or CORS, is a security feature of modern web browsers. It enables web browsers to negotiate which domains can make requests of external websites or services.

CORS is an important consideration when developing browser applications with the AWS SDK for JavaScript because most requests to resources are sent to an external domain, such as the endpoint for a web service. If your JavaScript environment enforces CORS security, you must configure CORS with the service.

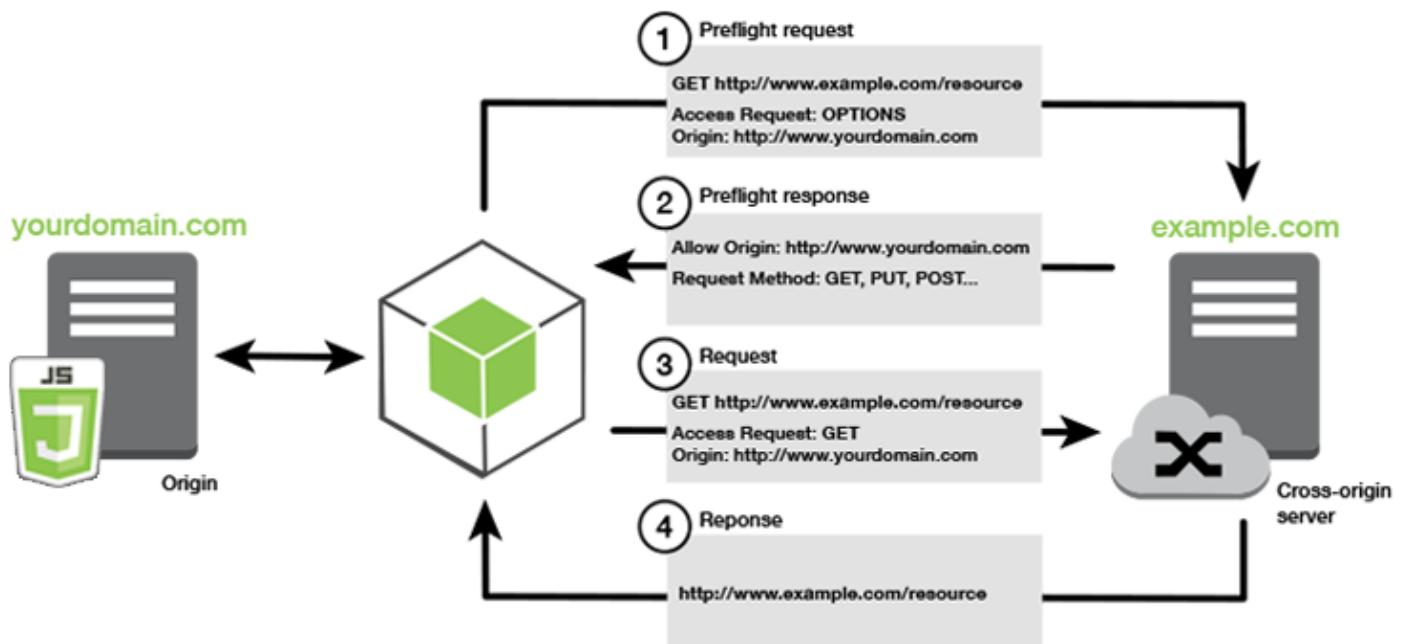
CORS determines whether to allow sharing of resources in a cross-origin request based on the following:

- The specific domain that makes the request
- The type of HTTP request being made (GET, PUT, POST, DELETE and so on)

How CORS works

In the simplest case, your browser script makes a GET request for a resource from a server in another domain. Depending on the CORS configuration of that server, if the request is from a domain that's authorized to submit GET requests, the cross-origin server responds by returning the requested resource.

If either the requesting domain or the type of HTTP request is not authorized, the request is denied. However, CORS makes it possible to preflight the request before actually submitting it. In this case, a preflight request is made in which the OPTIONS access request operation is sent. If the cross-origin server's CORS configuration grants access to the requesting domain, the server sends back a preflight response that lists all the HTTP request types that the requesting domain can make on the requested resource.



Is CORS configuration required?

Amazon S3 buckets require CORS configuration before you can perform operations on them. In some JavaScript environments CORS might not be enforced and therefore configuring CORS is unnecessary. For example, if you host your application from an Amazon S3 bucket and access resources from `*.s3.amazonaws.com` or some other specific endpoint, your requests won't access an external domain. Therefore, this configuration doesn't require CORS. In this case, CORS is still used for services other than Amazon S3.

Configure CORS for an Amazon S3 bucket

You can configure an Amazon S3 bucket to use CORS in the Amazon S3 console.

If you are configuring CORS in the AWS Web Services Management Console, you must use JSON to create a CORS configuration. The new AWS Web Services Management Console only supports JSON CORS configurations.

⚠ Important

In the new AWS Web Services Management Console, the CORS configuration must be JSON.

1. In the AWS Web Services Management Console, open the Amazon S3 console, find the bucket you want to configure and select its check box.
2. In the pane that opens, choose **Permissions**.
3. On the **Permission** tab, choose **CORS Configuration**.
4. Enter your CORS configuration in the **CORS Configuration Editor**, and then choose **Save**.

A CORS configuration is an XML file that contains a series of rules within a `<CORSRule>`. A configuration can have up to 100 rules. A rule is defined by one of the following tags:

- `<AllowedOrigin>` – Specifies domain origins that you allow to make cross-domain requests.
- `<AllowedMethod>` – Specifies a type of request you allow (GET, PUT, POST, DELETE, HEAD) in cross-domain requests.
- `<AllowedHeader>` – Specifies the headers allowed in a preflight request.

For example configurations, see [How do I configure CORS on my bucket?](#) in the *Amazon Simple Storage Service User Guide*.

CORS configuration example

The following CORS configuration example allows a user to view, add, remove, or update objects inside of a bucket from the domain `example.org`. However, we recommend that you scope the `<AllowedOrigin>` to the domain of your website. You can specify `"*"` to allow any origin.

Important

In the new S3 console, the CORS configuration must be JSON.

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>
  </CORSRule>
</CORSConfiguration>
```

```
<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"
    ]
  }
]
```

This configuration does not authorize the user to perform actions on the bucket. It enables the browser's security model to allow a request to Amazon S3. Permissions must be configured through bucket permissions or IAM role permissions.

You can use `ExposeHeader` to let the SDK read response headers returned from Amazon S3. For example, read the `ETag` header from a `PUT` or multipart upload, you need to include the `ExposeHeader` tag in your configuration, as shown in the previous example. The SDK can only access headers that are exposed through CORS configuration. If you set metadata on the object, values are returned as headers with the prefix `x-amz-meta-`, such as `x-amz-meta-my-custom-header`, and must also be exposed in the same way.

Bundle applications with webpack

The use of code modules by web applications in browser scripts or Node.js creates dependencies. These code modules can have dependencies of their own, resulting in a collection of interconnected modules that your application requires to function. To manage dependencies, you can use a module bundler like webpack.

The webpack module bundler parses your application code, searching for `import` or `require` statements, to create bundles that contain all the assets your application needs. This is so that the assets can be easily served through a webpage. The SDK for JavaScript can be included in webpack as one of the dependencies to include in the output bundle.

For more information about webpack, see the [webpack module bundler](#) on GitHub.

Install webpack

To install the webpack module bundler, you must first have npm, the Node.js package manager, installed. Type the following command to install the webpack CLI and JavaScript module.

```
npm install --save-dev webpack
```

To use the path module for working with file and directory paths, which is installed automatically with webpack, you might need to install the Node.js path-browserify package.

```
npm install --save-dev path-browserify
```

Configure webpack

By default, Webpack searches for a JavaScript file named `webpack.config.js` in your project's root directory. This file specifies your configuration options. The following is an example of a `webpack.config.js` configuration file for WebPack version 5.0.0 and later.

Note

Webpack configuration requirements vary depending on the version of Webpack you install. For more information, see the [Webpack documentation](#).

```
// 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'
  },
  // Enable WebPack to use the 'path' package.
  resolve: {
    fallback: { path: require.resolve("path-browserify")}
  }
  /**
   * In Webpack version v2.0.0 and earlier, you must tell
   * webpack how to use "json-loader" to load 'json' files.
   * To do this Enter 'npm --save-dev install json-loader' at the
   * command line to install the "json-loader" package, and include the
   * following entry in your webpack.config.js.
   * module: {
     rules: [{test: /\.json$/, use: use: "json-loader"}]
   }
  **/
};
```

In this example, `browser.js` is specified as the *entry point*. The *entry point* is the file webpack uses to begin searching for imported modules. The file name of the output is specified as `bundle.js`. This output file will contain all the JavaScript the application needs to run. If the code specified in the entry point imports or requires other modules, such as the SDK for JavaScript, that code is bundled without needing to specify it in the configuration.

Run webpack

To build an application to use webpack, add the following to the `scripts` object in your `package.json` file.

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

The following is an example `package.json` file that demonstrates adding webpack.

```
{
```

```
"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/client-iam": "^3.32.0",
  "@aws-sdk/client-s3": "^3.32.0"
},
"devDependencies": {
  "webpack": "^5.0.0"
}
}
```

To build your application, enter the following command.

```
npm run build
```

The webpack module bundler then generates the JavaScript file you specified in your project's root directory.

Use the webpack bundle

To use the bundle in a browser script, you can incorporate the bundle using a `<script>` tag, as shown in the following example.

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

Bundle for Node.js

You can use `webpack` to generate bundles that run in Node.js by specifying `node` as a target in the configuration.

```
target: "node"
```

This is useful when running a Node.js application in an environment where disk space is limited. Here is an example `webpack.config.js` configuration with Node.js specified as the output target.

```
// 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'
  },
  // Let webpack know to generate a Node.js bundle.
  target: "node",
  // Enable WebPack to use the 'path' package.
  resolve:{
  fallback: { path: require.resolve("path-browserify")}
  /**
   * In Webpack version v2.0.0 and earlier, you must tell
   * webpack how to use "json-loader" to load 'json' files.
   * To do this Enter 'npm --save-dev install json-loader' at the
   * command line to install the "json-loader" package, and include the
   * following entry in your webpack.config.js.
   */
  module: {
    rules: [{test: /\.json$/, use: use: "json-loader"}]
  }
  /**/
};
```


Work with AWS services in the SDK for JavaScript

The AWS SDK for JavaScript v3 provides access to services that it supports through a collection of client classes. From these client classes, you create service interface objects, commonly called *service objects*. Each supported AWS service has one or more client classes that offer low-level APIs for using service features and resources. For example, Amazon DynamoDB APIs are available through the DynamoDB class.

The services exposed through the SDK for JavaScript follow the request-response pattern to exchange messages with calling applications. In this pattern, the code invoking a service submits an HTTP/HTTPS request to an endpoint for the service. The request contains parameters needed to successfully invoke the specific feature being called. The service that is invoked generates a response that is sent back to the requestor. The response contains data if the operation was successful or error information if the operation was unsuccessful.

Invoking an AWS service includes the full request and response lifecycle of an operation on a service object, including any retries that are attempted. A request contains zero or more properties as JSON parameters. The response is encapsulated in an object related to the operation, and is returned to the requestor through one of several techniques, such as a callback function or a JavaScript promise.

Topics

- [Create and call service objects](#)
- [Call services asynchronously](#)
- [Create service client requests](#)
- [Handle service client responses](#)
- [Work with JSON](#)
- [SDK for JavaScript code examples](#)

Create and call service objects

The JavaScript API supports most available AWS services. Each service in the JavaScript API provides a client class with a `send` method that you use to to invoke every API the service supports. For more information about service classes, operations, and parameters in the JavaScript API, see the [API Reference](#).

When using the SDK in Node.js, you add the SDK package for each service you need to your application using `import`, which provides support for all current services. The following example creates an Amazon S3 service object in the `us-west-1` Region.

```
// Import the Amazon S3 service client
import { S3Client } from "@aws-sdk/client-s3";
// Create an S3 client in the us-west-1 Region
const s3Client = new S3Client({
  region: "us-west-1"
});
```

Specify service object parameters

When calling a method of a service object, pass parameters in JSON as required by the API. For example, in Amazon S3, to get an object for a specified bucket and key, pass the following parameters to the `GetObjectCommand` method from the `S3Client`. For more information about passing JSON parameters, see [Work with JSON](#).

```
s3Client.send(new GetObjectCommand({Bucket: 'bucketName', Key: 'keyName'}));
```

For more information about Amazon S3 parameters, see [@aws-sdk/client-s3](#) in the API Reference.

Call services asynchronously

All requests made through the SDK are asynchronous. This is important to keep in mind when writing browser scripts. JavaScript running in a web browser typically has just a single execution thread. After making an asynchronous call to an AWS service, the browser script continues running and in the process can try to execute code that depends on that asynchronous result before it returns.

Making asynchronous calls to an AWS service includes managing those calls so your code doesn't try to use data before the data is available. The topics in this section explain the need to manage asynchronous calls and detail different techniques you can use to manage them.

Although you can use any of these techniques to manage asynchronous calls, we recommend that you use `async/await` for all new code.

`async/await`

We recommend that you use this technique as it is the default behavior in V3.

promise

Use this technique in browsers that do not support `async/await`.

callback

Avoid using callbacks except in very simple cases. However, you might find it useful for migration scenarios.

Topics

- [Manage asynchronous calls](#)
- [Use `async/await`](#)
- [Use JavaScript promises](#)
- [Use an anonymous callback function](#)

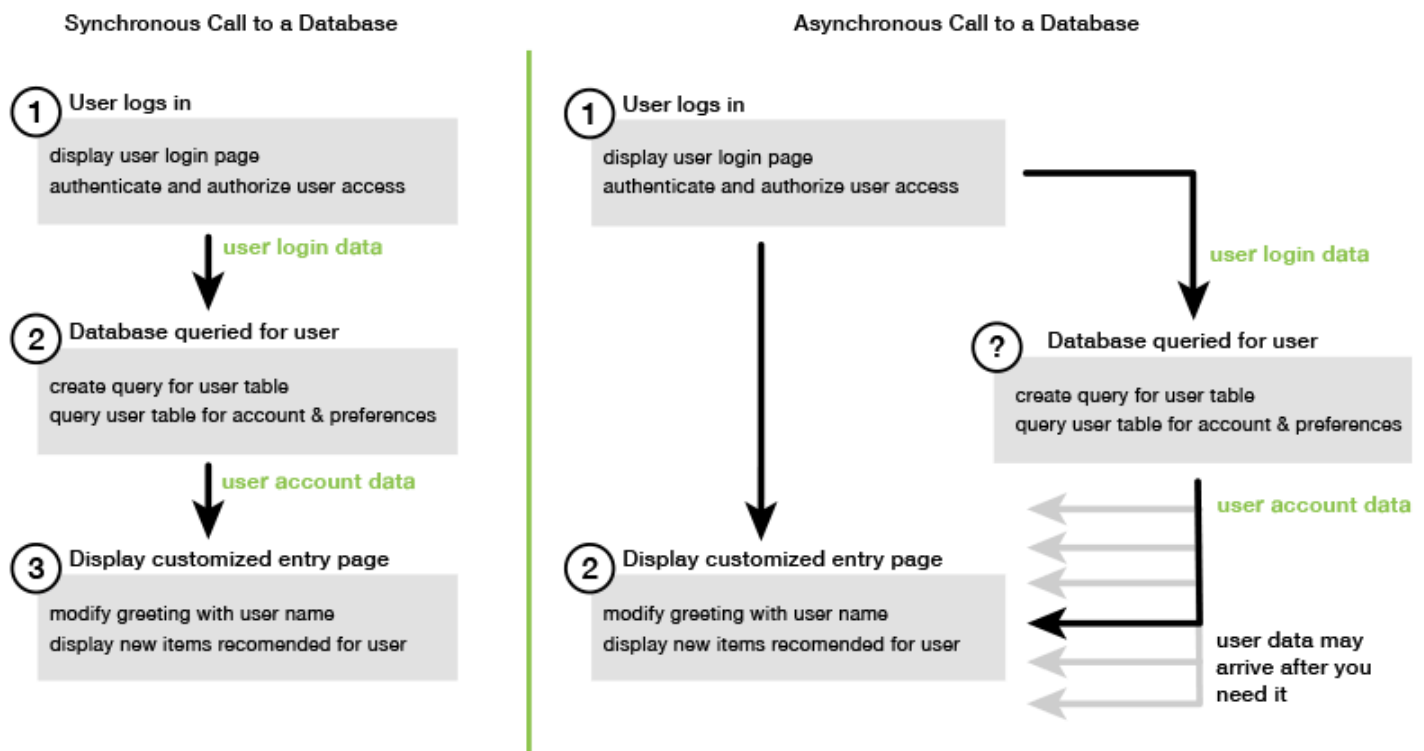
Manage asynchronous calls

For example, the home page of an e-commerce website lets returning customers sign in. Part of the benefit for customers who sign in is that, after signing in, the site then customizes itself to their particular preferences. To make this happen:

1. The customer must log in and be validated with their sign-in credentials.
2. The customer's preferences are requested from a customer database.
3. The database provides the customer's preferences that are used to customize the site before the page loads.

If these tasks execute synchronously, then each must finish before the next can start. The webpage would be unable to finish loading until the customer preferences return from the database. However, after the database query is sent to the server, receipt of the customer data can be delayed or even fail due to network bottlenecks, exceptionally high database traffic, or a poor mobile device connection.

To keep the website from freezing under those conditions, call the database asynchronously. After the database call executes, sending your asynchronous request, your code continues to execute as expected. If you don't properly manage the response of an asynchronous call, your code can attempt to use information it expects back from the database when that data isn't available yet.



Use async/await

Rather than using promises, you should consider using `async/await`. Async functions are simpler and take less boilerplate than using promises. `Await` can only be used in an async function to asynchronously wait for a value.

The following example uses `async/await` to list all of your Amazon DynamoDB tables in `us-west-2`.

Note

For this example to run:

- Install the AWS SDK for JavaScript DynamoDB client by entering `npm install @aws-sdk/client-dynamodb` in the command line of your project.
- Ensure you have configured your AWS credentials correctly. For more information, see [Set credentials](#).

```
import {
```

```
DynamoDBClient,
ListTablesCommand
} from "@aws-sdk/client-dynamodb";
(async function () {
  const dbClient = new DynamoDBClient({ region: "us-west-2" });
  const command = new ListTablesCommand({});

  try {
    const results = await dbClient.send(command);
    console.log(results.TableNames.join('\n'));
  } catch (err) {
    console.error(err)
  }
})();
```

Note

Not all browsers support `async/await`. See [Async functions](#) for a list of browsers with `async/await` support.

Use JavaScript promises

Use the service client's AWS SDK for JavaScript v3 method (`ListTablesCommand`) to make the service call and manage asynchronous flow instead of using callbacks. The following example shows how to get the names of your Amazon DynamoDB tables in `us-west-2`.

```
import {
  DynamoDBClient,
  ListTablesCommand
} from "@aws-sdk/client-dynamodb";
const dbClient = new DynamoDBClient({ region: 'us-west-2' });

dbClient.listtables(new ListTablesCommand({}))
  .then(response => {
    console.log(response.TableNames.join('\n'));
  })
  .catch((error) => {
    console.error(error);
  });
```

Coordinate multiple promises

In some situations, your code must make multiple asynchronous calls that require action only when they have all returned successfully. If you manage those individual asynchronous method calls with promises, you can create an additional promise that uses the `all` method.

This method fulfills this umbrella promise if and when the array of promises that you pass into the method are fulfilled. The callback function is passed an array of the values of the promises passed to the `all` method.

In the following example, an AWS Lambda function must make three asynchronous calls to Amazon DynamoDB but can only complete after the promises for each call are fulfilled.

```
const values = await Promise.all([firstPromise, secondPromise, thirdPromise]);

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 values;
```

Browser and Node.js support for promises

Support for native JavaScript promises (ECMAScript 2015) depends on the JavaScript engine and version in which your code executes. To help determine the support for JavaScript promises in each environment where your code needs to run, see the [ECMAScript compatibility table](#) on GitHub.

Use an anonymous callback function

Each service object method can accept an anonymous callback function as the last parameter. The signature of this callback function is as follows.

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

This callback function executes when either a successful response or error data returns. If the method call succeeds, the contents of the response are available to the callback function in the `data` parameter. If the call doesn't succeed, the details about the failure are provided in the `error` parameter.

Typically the code inside the callback function tests for an error, which it processes if one is returned. If an error is not returned, the code then retrieves the data in the response from the data parameter. The basic form of the callback function looks like this example.

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

In the previous example, the details of either the error or the returned data are logged to the console. Here is an example that shows a callback function passed as part of calling a method on a service object.

```
ec2.describeInstances(function(error, data) {
  if (error) {
    console.log(error); // an error occurred
  } else {
    console.log(data); // request succeeded
  }
});
```

Create service client requests

Making requests to AWS service clients is straightforward. Version 3 (V3) of the SDK for JavaScript enables you to send requests.

Note

You can also perform operations using version 2 (V2) commands when using the V3 of the SDK for JavaScript. For more information, see [Using v2 commands](#).

To send a request:

1. Initialize a client object with the desired configuration, such as a specific AWS Region.

2. (Optional) Create a request JSON object with the values for the request, such as the name of a specific Amazon S3 bucket. You can examine the parameters for the request by looking at the API Reference topic for the interface with the name associated with the client method. For example, if you use the *AbcCommand* client method, the request interface is *AbcInput*.
3. Initialize a service command, optionally, with the request object as input.
4. Call `send` on the client with the command object as input.

For example, to list your Amazon DynamoDB tables in `us-west-2`, you can do it with `async/await`.

```
import {
  DynamoDBClient,
  ListTablesCommand
} from "@aws-sdk/client-dynamodb";

(async function () {
  const dbClient = new DynamoDBClient({ region: 'us-west-2' });
  const command = new ListTablesCommand({});

  try {
    const results = await dbClient.send(command);
    console.log(results.TableNames.join('\n'));
  } catch (err) {
    console.error(err);
  }
})();
```

Handle service client responses

After a service client method has been called, it returns a response object instance of an interface with the name associated with the client method. For example, if you use the *AbcCommand* client method, the response object is of *AbcResponse* (interface) type.

Access data returned in the response

The response object contains the data, as properties, returned by the service request.

In [Create service client requests](#), the `ListTablesCommand` command returned the table names in the `TableNames` property of the response.

Access error information

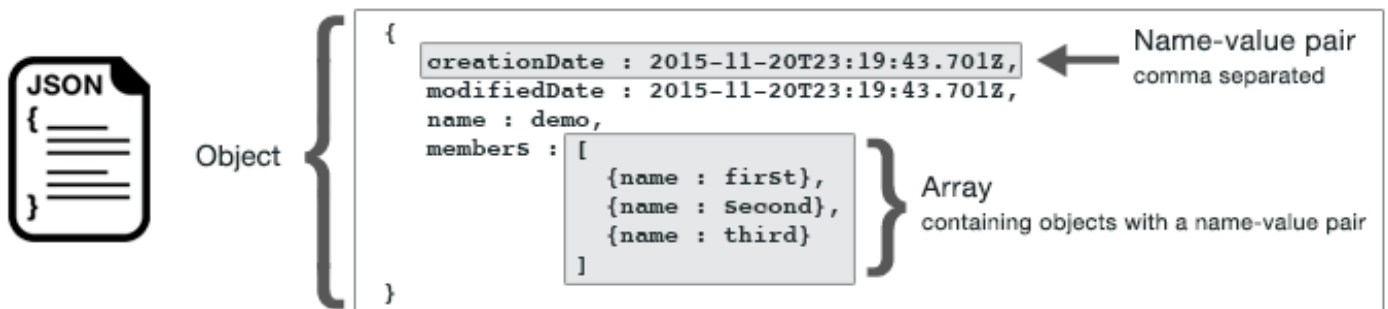
If a command fails, it throws an exception. The following code snippet shows a way of handling a service exception.

```
try {
  await client.send(someCommand);
} catch (e) {
  if (e.name === "InvalidSignatureException") {
    // Handle InvalidSignatureException
  } else if (e.name === "ResourceNotFoundException") {
    // Handle ResourceNotFoundException
  } else if (e.name === "FooServiceException") {
    // Handle all other server-side exceptions from Foo service
  } else {
    // Handle errors from SDK
  }
}
```

Work with JSON

JSON is a format for data exchange that is both human-readable and machine-readable. Although the name JSON is an acronym for *JavaScript Object Notation*, the format of JSON is independent of any programming language.

The AWS SDK for JavaScript uses JSON to send data to service objects when making requests and receives data from service objects as JSON. For more information about JSON, see [json.org](https://www.json.org).



JSON represents data in two ways:

- As an *object*, which is an unordered collection of name-value pairs. An object is defined within left ({) and right (}) braces. Each name-value pair begins with the name, followed by a colon, followed by the value. Name-value pairs are comma separated.
- As an *array*, which is an ordered collection of values. An array is defined within left ([) and right (]) brackets. Items in the array are comma separated.

Here is an example of a JSON object that contains an array of objects in which the objects represent cards in a card game. Each card is defined by two name-value pairs, one that specifies a unique value to identify that card and another that specifies a URL that points to the corresponding card image.

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

JSON as service object parameters

Here is an example of simple JSON used to define the parameters of a call to an AWS Lambda service object.

```
const params = {  
  FunctionName : funcName,  
  Payload : JSON.stringify(payload),  
  LogType : LogType.Tail,  
};
```

The `params` object is defined by three name-value pairs, separated by commas within the left and right braces. When providing parameters to a service object method call, the names are determined by the parameter names for the service object method you plan to call. When invoking a Lambda function, `FunctionName`, `Payload`, and `LogType` are the parameters used to call the `invoke` method on a Lambda service object.

When passing parameters to a service object method call, provide the JSON object to the method call, as shown in the following example of invoking a Lambda function.

```
const invoke = async (funcName, payload) => {
  const client = new LambdaClient({});
  const command = new InvokeCommand({
    FunctionName: funcName,
    Payload: JSON.stringify(payload),
    LogType: LogType.Tail,
  });

  const { Payload, LogResult } = await client.send(command);
  const result = Buffer.from(Payload).toString();
  const logs = Buffer.from(LogResult, "base64").toString();
  return { logs, result };
};
```

SDK for JavaScript code examples

The topics in this section contain examples of how to use the AWS SDK for JavaScript with the APIs of various services to carry out common tasks.

Find the source code for these examples and others in the [AWS Code Examples Repository on GitHub](#). To propose a new code example for the AWS documentation team to consider producing, create a request. The team is looking to produce code examples that cover broader scenarios and use cases, versus simple code snippets that cover only individual API calls. For instructions, see the *Authoring code* section in the [contributing guidelines on GitHub](#).

Important

These examples use ECMAScript6 import/export syntax.

- This require Node.js version 14.17 or higher. To download and install the latest version of Node.js, see [Node.js downloads](#).
- If you prefer to use CommonJS syntax, see [JavaScript ES6/CommonJS syntax](#) for conversion guidelines.

Topics

- [JavaScript ES6/CommonJS syntax](#)
- [Amazon DynamoDB examples](#)

- [AWS Elemental MediaConvert examples](#)
- [AWS Lambda examples](#)
- [Amazon Lex examples](#)
- [Amazon Polly examples](#)
- [Amazon Redshift examples](#)
- [Amazon Simple Email Service examples](#)
- [Amazon Simple Notification Service Examples](#)
- [Amazon Transcribe examples](#)
- [Setting up Node.js on an Amazon EC2 instance](#)
- [Build an app to submit data to DynamoDB](#)
- [Invoking Lambda with API Gateway](#)
- [Creating scheduled events to execute AWS Lambda functions](#)
- [Building an Amazon Lex chatbot](#)
- [Creating an example messaging application](#)

JavaScript ES6/CommonJS syntax

The AWS SDK for JavaScript code examples are written in ECMAScript 6 (ES6). ES6 brings new syntax and new features to make your code more modern and readable, and do more.

ES6 requires you use Node.js version 13.x or higher. To download and install the latest version of Node.js, see [Node.js downloads](#). However, if you prefer, you can convert any of our examples to CommonJS syntax using the following guidelines:

- Remove "type" : "module" from the package . json in your project environment.
- Convert all ES6 import statements to CommonJS require statements. For example, convert:

```
import { CreateBucketCommand } from "@aws-sdk/client-s3";
import { s3 } from "../libs/s3Client.js";
```

To its CommonJS equivalent:

```
const { CreateBucketCommand } = require("@aws-sdk/client-s3");
```

```
const { s3 } = require("../libs/s3Client.js");
```

- Convert all ES6 export statements to CommonJS `module.exports` statements. For example, convert:

```
export {s3}
```

To its CommonJS equivalent:

```
module.exports = {s3}
```

The following example demonstrates the code example for creating an Amazon S3 bucket in both ES6 and CommonJS.

ES6

libs/s3Client.js

```
// Create service client module using ES6 syntax.
import { S3Client } from "@aws-sdk/client-s3";
// Set the AWS region
const REGION = "eu-west-1"; //e.g. "us-east-1"
// Create Amazon S3 service object.
const s3 = new S3Client({ region: REGION });
// Export 's3' constant.
export {s3};
```

s3_createbucket.js

```
// Get service clients module and commands using ES6 syntax.
import { CreateBucketCommand } from "@aws-sdk/client-s3";
import { s3 } from "../libs/s3Client.js";

// Get service clients module and commands using CommonJS syntax.
// const { CreateBucketCommand } = require("@aws-sdk/client-s3");
// const { s3 } = require("../libs/s3Client.js");
```

```
// Set the bucket parameters
const bucketParams = { Bucket: "BUCKET_NAME" };

// Create the Amazon S3 bucket.
const run = async () => {
  try {
    const data = await s3.send(new CreateBucketCommand(bucketParams));
    console.log("Success", data.Location);
    return data;
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

CommonJS

libs/s3Client.js

```
// Create service client module using CommonJS syntax.
const { S3Client } = require("@aws-sdk/client-s3");
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create Amazon S3 service object.
const s3 = new S3Client({ region: REGION });
// Export 's3' constant.
module.exports = {s3};
```

s3_createbucket.js

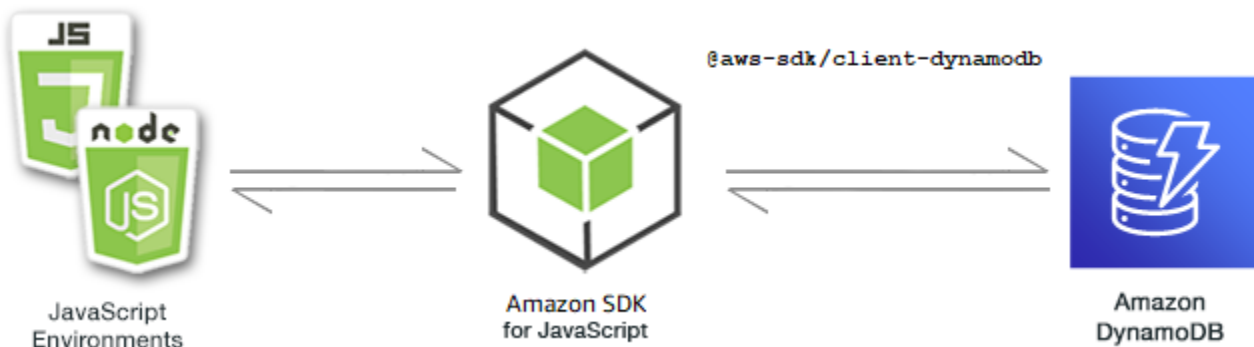
```
// Get service clients module and commands using CommonJS syntax.
const { CreateBucketCommand } = require("@aws-sdk/client-s3");
const { s3 } = require("../libs/s3Client.js");

// Set the bucket parameters
const bucketParams = { Bucket: "BUCKET_NAME" };
```

```
// Create the Amazon S3 bucket.
const run = async () => {
  try {
    const data = await s3.send(new CreateBucketCommand(bucketParams));
    console.log("Success", data.Location);
    return data;
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

Amazon DynamoDB examples

Amazon DynamoDB is a fully managed NoSQL cloud database that supports both document and key-value store models. You create schemaless tables for data without the need to provision or maintain dedicated database servers.



The JavaScript API for DynamoDB is exposed through the `DynamoDB`, `DynamoDBStreams`, and `DynamoDB.DocumentClient` client classes. For more information about using the DynamoDB client classes, see [Class: DynamoDB](#), [Class: DynamoDBStreams](#), and [Class: DynamoDB utility](#) in the API Reference.

Topics

- [Creating and using tables in DynamoDB](#)
- [Reading and writing a single item in DynamoDB](#)
- [Reading and writing items in batch in DynamoDB](#)
- [Querying and scanning a DynamoDB table](#)

- [Using the DynamoDB Document Client](#)

Creating and using tables in DynamoDB



This Node.js code example shows:

- How to create and manage tables used to store and retrieve data from DynamoDB.

The scenario

Similar to other database systems, DynamoDB stores data in tables. A DynamoDB table is a collection of data that's organized into items that are analogous to rows. To store or access data in DynamoDB, you create and work with tables.

In this example, you use a series of Node.js modules to perform basic operations with a DynamoDB table. The code uses the SDK for JavaScript to create and work with tables by using these methods of the DynamoDB client class:

- [CreateTableCommand](#)
- [ListTablesCommand](#)
- [DescribeTableCommand](#)
- [DeleteTableCommand](#)

Prerequisite tasks

To set up and run this example, first complete these tasks:

- Set up the project environment to run these Node.js examples, and install the required AWS SDK for JavaScript and third-party modules. Follow the instructions on [GitHub](#).
- Install SDK for JavaScript DynamoDB client. For more information, see [What's new in Version 3](#).
- Create a shared configurations file with your user credentials. For more information about providing a shared credentials file, see [Shared config and credentials files](#) in the *AWS SDKs and Tools Reference Guide*.

⚠ Important

These examples use ECMAScript6 (ES6). This requires Node.js version 13.x or higher. To download and install the latest version of Node.js, see [Node.js downloads](#).

However, if you prefer to use CommonJS syntax, please refer to [JavaScript ES6/CommonJS syntax](#).

ℹ Note

For information about the data types used in these examples, see [Supported data types and naming rules in Amazon DynamoDB](#).

Creating a table

Create a Node.js module with the file name `create-table.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. To access DynamoDB, create a DynamoDB client service object. Create a JSON object containing the parameters needed to create a table, which in this example includes the name and data type for each attribute, the key schema, the name of the table, and the units of throughput to provision. Call the `CreateTableCommand` method of the DynamoDB service object.

```
import { CreateTableCommand, DynamoDBClient } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({});

export const main = async () => {
  const command = new CreateTableCommand({
    TableName: "EspressoDrinks",
    // For more information about data types,
    // see https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
    // HowItWorks.NamingRulesDataTypes.html#HowItWorks.DataTypes and
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
    // Programming.LowLevelAPI.html#Programming.LowLevelAPI.DataTypeDescriptors
    AttributeDefinitions: [
      {
        AttributeName: "DrinkName",
        AttributeType: "S",
      },
    ],
  });
};
```

```
    ],
    KeySchema: [
      {
        AttributeName: "DrinkName",
        KeyType: "HASH",
      },
    ],
    ProvisionedThroughput: {
      ReadCapacityUnits: 1,
      WriteCapacityUnits: 1,
    },
  });

  const response = await client.send(command);
  console.log(response);
  return response;
};
```

To run the example, enter the following at the command prompt.

```
node create-table.js
```

This example code can be found [here on GitHub](#).

Listing your tables

Create a Node.js module with the file name `list-tables.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. To access DynamoDB, create a DynamoDB client service object. Create a JSON object containing the parameters needed to list your tables, which in this example limits the number of tables listed to 10. Call the `ListTablesCommand` method of the DynamoDB service object.

```
import { ListTablesCommand, DynamoDBClient } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({});

export const main = async () => {
  const command = new ListTablesCommand({});

  const response = await client.send(command);
  console.log(response);
  return response;
};
```

```
};
```

To run the example, enter the following at the command prompt.

```
node list-tables.js
```

This example code can be found [here on GitHub](#).

Describing a table

Create a Node.js module with the file name `describe-table.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. To access DynamoDB, create a DynamoDB client service object. Create a JSON object containing the parameters needed to describe a `DescribeTableCommand` method of the DynamoDB service object.

```
import { DescribeTableCommand, DynamoDBClient } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({});

export const main = async () => {
  const command = new DescribeTableCommand({
    TableName: "Pastries",
  });

  const response = await client.send(command);
  console.log(`TABLE NAME: ${response.Table.TableName}`);
  console.log(`TABLE ITEM COUNT: ${response.Table.ItemCount}`);
  return response;
};
```

To run the example, enter the following at the command prompt.

```
node describe-table.js
```

This example code can be found [here on GitHub](#).

Deleting a table

Create a Node.js module with the file name `delete-table.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. To access DynamoDB, create a DynamoDB client service object. Create a JSON object containing the parameters needed

to delete a table, which in this example includes the name of the table provided as a command-line parameter. Call the `DeleteTableCommand` method of the DynamoDB service object.

```
import { DeleteTableCommand, DynamoDBClient } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({});

export const main = async () => {
  const command = new DeleteTableCommand({
    TableName: "DecafCoffees",
  });

  const response = await client.send(command);
  console.log(response);
  return response;
};
```

To run the example, enter the following at the command prompt.

```
node delete-table.js
```

This example code can be found [here on GitHub](#).

Reading and writing a single item in DynamoDB



This Node.js code example shows:

- How to add an item in a DynamoDB table.
- How to retrieve an item in a DynamoDB table.
- How to delete an item in a DynamoDB table.

The scenario

In this example, you use a series of Node.js modules to read and write one item in a DynamoDB table by using these methods of the DynamoDB client class:

- [PutItemCommand](#)
- [UpdateItemCommand](#)
- [GetItemCommand](#)
- [DeleteItemCommand](#)

Prerequisite tasks

To set up and run this example, first complete these tasks:

- Set up the project environment to run these Node.js examples, and install the required AWS SDK for JavaScript and third-party modules. Follow the instructions on [GitHub](#).
- Create a shared configurations file with your user credentials. For more information about providing a shared credentials file, see [Shared config and credentials files](#) in the *AWS SDKs and Tools Reference Guide*.
- Create a DynamoDB table whose items you can access. For more information about creating a DynamoDB table, see [Creating and using tables in DynamoDB](#).

Important

These examples use ECMAScript6 (ES6). This requires Node.js version 13.x or higher. To download and install the latest version of Node.js, see [Node.js downloads](#).

However, if you prefer to use CommonJS syntax, please refer to [JavaScript ES6/CommonJS syntax](#).

Note

For information about the data types used in these examples, see [Supported data types and naming rules in Amazon DynamoDB](#).

Writing an item

Create a Node.js module with the file name `put-item.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. To access DynamoDB, create a DynamoDB client service object. Create a JSON object containing the parameters needed

to add an item, which in this example includes the name of the table and a map that defines the attributes to set and the values for each attribute. Call the `PutItemCommand` method of the DynamoDB client service object.

```
import { PutItemCommand, DynamoDBClient } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({});

export const main = async () => {
  const command = new PutItemCommand({
    TableName: "Cookies",
    // For more information about data types,
    // see https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
    // HowItWorks.NamingRulesDataTypes.html#HowItWorks.DataTypes and
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
    // Programming.LowLevelAPI.html#Programming.LowLevelAPI.DataTypeDescriptors
    Item: {
      Flavor: { S: "Chocolate Chip" },
      Variants: { SS: ["White Chocolate Chip", "Chocolate Chunk" ] },
    },
  });

  const response = await client.send(command);
  console.log(response);
  return response;
};
```

To run the example, enter the following at the command prompt.

```
node put-item.js
```

This example code can be found [here on GitHub](#).

Update an item

Create a Node.js module with the file name `update-item.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. To access DynamoDB, create a DynamoDB client service object. Create a JSON object containing the parameters needed to add an item, which in this example includes the name of the table, the key to update, and the date expression that maps the new attribute names, and values for each new attribute. Call the `UpdateItemCommand` method of the DynamoDB client service object.

```
import { UpdateItemCommand, DynamoDBClient } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({});

export const main = async () => {
  const command = new UpdateItemCommand({
    TableName: "IceCreams",
    // For more information about data types,
    // see https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
    // HowItWorks.NamingRulesDataTypes.html#HowItWorks.DataTypes and
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
    // Programming.LowLevelAPI.html#Programming.LowLevelAPI.DataTypeDescriptors
    Key: {
      Flavor: { S: "Vanilla" },
    },
    UpdateExpression: "set HasChunks = :chunks",
    ExpressionAttributeValues: {
      ":chunks": { BOOL: "false" },
    },
    ReturnValues: "ALL_NEW",
  });

  const response = await client.send(command);
  console.log(response);
  return response;
};
```

To run the example, enter the following at the command prompt.

```
node update-item.js
```

This example code can be found [here on GitHub](#).

Getting an item

Create a Node.js module with the file name `get-item.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. To access DynamoDB, create a DynamoDB client service object. To identify the item to get, you must provide the value of the primary key for that item in the table. By default, the `GetItemCommand` method returns all the attribute values defined for the item. To get only a subset of all possible attribute values, specify a projection expression.

Create a JSON object containing the parameters needed to get an item, which in this example includes the name of the table, the name and value of the key for the item you're getting, and a projection expression that identifies the item attribute you want to retrieve. Call the `GetItemCommand` method of the DynamoDB client service object.

The following code example retrieves an item from a table with a primary key composed of only a partition key and not of both a partition and sort key. If the table has a primary key composed of a partition key and a sort key, you must also specify the sort key name and attribute.

```
import { GetItemCommand, DynamoDBClient } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({});

export const main = async () => {
  const command = new GetItemCommand({
    TableName: "CafeTreats",
    // For more information about data types,
    // see https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
    // HowItWorks.NamingRulesDataTypes.html#HowItWorks.DataTypes and
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
    // Programming.LowLevelAPI.html#Programming.LowLevelAPI.DataTypeDescriptors
    Key: {
      TreatId: { N: "101" },
    },
  });

  const response = await client.send(command);
  console.log(response);
  return response;
};
```

To run the example, enter the following at the command prompt.

```
node get-item.js
```

This example code can be found [here on GitHub](#).

Deleting an item

Create a Node.js module with the file name `delete-item.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. To access DynamoDB,

create a DynamoDB client service object. Create a JSON object containing the parameters needed to delete an item, which in this example includes the name of the table and both the key name and value for the item you're deleting. Call the `DeleteItemCommand` method of the DynamoDB client service object.

```
import { DeleteItemCommand, DynamoDBClient } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({});

export const main = async () => {
  const command = new DeleteItemCommand({
    TableName: "Drinks",
    // For more information about data types,
    // see https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
    // HowItWorks.NamingRulesDataTypes.html#HowItWorks.DataTypes and
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
    // Programming.LowLevelAPI.html#Programming.LowLevelAPI.DataTypeDescriptors
    Key: {
      Name: { S: "Pumpkin Spice Latte" },
    },
  });

  const response = await client.send(command);
  console.log(response);
  return response;
};
```

To run the example, enter the following at the command prompt.

```
node delete-item.js
```

This example code can be found [here on GitHub](#).

Reading and writing items in batch in DynamoDB



This Node.js code example shows:

- How to read and write batches of items in a DynamoDB table.

The scenario

In this example, you use a series of Node.js modules to put a batch of items in a DynamoDB table and read a batch of items. The code uses the SDK for JavaScript to perform batch read and write operations using these methods of the DynamoDB client class:

- [BatchGetItemCommand](#)
- [BatchWriteItemCommand](#)

Prerequisite tasks

To set up and run this example, first complete these tasks:

- Set up the project environment to run these Node TypeScript examples, and install the required AWS SDK for JavaScript and third-party modules. Follow the instructions on [GitHub](#).
- Create a shared configurations file with your user credentials. For more information about providing a shared credentials file, see [Shared config and credentials files](#) in the *AWS SDKs and Tools Reference Guide*.
- Create a DynamoDB table whose items you can access. For more information about creating a DynamoDB table, see [Creating and using tables in DynamoDB](#).

Important

These examples use ECMAScript6 (ES6). This requires Node.js version 13.x or higher. To download and install the latest version of Node.js, see [Node.js downloads](#). However, if you prefer to use CommonJS syntax, please refer to [JavaScript ES6/CommonJS syntax](#).

Note

For information about the data types used in these examples, see [Supported data types and naming rules in Amazon DynamoDB](#).

Reading items in a batch

Create a Node.js module with the file name `batch-get-item.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. To access DynamoDB, create a DynamoDB client service object. Create a JSON object containing the parameters needed to get a batch of items, which in this example includes the name of one or more tables from which to read, the values of keys to read in each table, and the projection expression that specifies the attributes to return. Call the `BatchGetItemCommand` method of the DynamoDB service object.

```
import { BatchGetItemCommand, DynamoDBClient } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({});

export const main = async () => {
  const command = new BatchGetItemCommand({
    RequestItems: {
      // Each key in this object is the name of a table. This example refers
      // to a PageAnalytics table.
      PageAnalytics: {
        // Each entry in Keys is an object that specifies a primary key.
        Keys: [
          {
            // "PageName" is the partition key (simple primary key).
            // "S" specifies a string as the data type for the value "Home".
            // For more information about data types,
            // see https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.NamingRulesDataTypes.html#HowItWorks.DataTypes and
            // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Programming.LowLevelAPI.html#Programming.LowLevelAPI.DataTypeDescriptors
            PageName: { S: "Home" },
          },
          {
            PageName: { S: "About" },
          },
        ],
      },
      // Only return the "PageName" and "PageViews" attributes.
      ProjectionExpression: "PageName, PageViews",
    },
  });

  const response = await client.send(command);
  console.log(response.Responses["PageAnalytics"]);
};
```

```
    return response;
  };
```

To run the example, enter the following at the command prompt.

```
node batch-get-item.js
```

This example code can be found [here on GitHub](#).

Writing items in a batch

Create a Node.js module with the file name `batch-write-item.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. To access DynamoDB, create a DynamoDB client service object. Create a JSON object containing the parameters needed to get a batch of items, which in this example includes the table into which you want to write items, the keys you want to write for each item, and the attributes along with their values. Call the `BatchWriteItemCommand` method of the DynamoDB service object.

```
import {
  BatchWriteItemCommand,
  DynamoDBClient,
} from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({});

export const main = async () => {
  const command = new BatchWriteItemCommand({
    RequestItems: {
      // Each key in this object is the name of a table. This example refers
      // to a Coffees table.
      Coffees: [
        // Each entry in Coffees is an object that defines either a PutRequest or
        DeleteRequest.
        {
          // Each PutRequest object defines one item to be inserted into the table.
          PutRequest: {
            // The keys of Item are attribute names. Each attribute value is an object
            with a data type and value.
            // For more information about data types,
            // see https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
            HowItWorks.NamingRulesDataTypes.html#HowItWorks.DataTypes
```

```
        Item: {
          Name: { S: "Donkey Kick" },
          Process: { S: "Wet-Hulled" },
          Flavors: { SS: ["Earth", "Syrup", "Spice"] },
        },
      },
    ],
  },
  {
    PutRequest: {
      Item: {
        Name: { S: "Flora Ethiopia" },
        Process: { S: "Washed" },
        Flavors: { SS: ["Stone Fruit", "Toasted Almond", "Delicate"] },
      },
    },
  },
],
},
});

const response = await client.send(command);
console.log(response);
return response;
};
```

```
const response = await client.send(command);
console.log(response);
return response;
};
```

To run the example, enter the following at the command prompt.

```
node batch-write-item.js
```

This example code can be found [here on GitHub](#).

Querying and scanning a DynamoDB table



This Node.js code example shows:

- How to query and scan a DynamoDB table for items.

The scenario

Querying finds items in a table or a secondary index using only primary key attribute values. You must provide a partition key name and a value for which to search. You can also provide a sort key name and value, and use a comparison operator to refine the search results. Scanning finds items by checking every item in the specified table.

In this example, you use a series of Node.js modules to identify one or more items you want to retrieve from a DynamoDB table. The code uses the SDK for JavaScript to query and scan tables using these methods of the DynamoDB client class:

- [QueryCommand](#)
- [ScanCommand](#)

Prerequisite tasks

To set up and run this example, first complete these tasks:

- Set up the project environment to run these Node.js examples, and install the required AWS SDK for JavaScript and third-party modules. Follow the instructions on [GitHub](#).
- Create a shared configurations file with your user credentials. For more information about providing a shared credentials file, see [Shared config and credentials files](#) in the *AWS SDKs and Tools Reference Guide*.
- Create a DynamoDB table whose items you can access. For more information about creating a DynamoDB table, see [Creating and using tables in DynamoDB](#).

Important

These examples use ECMAScript6 (ES6). This requires Node.js version 13.x or higher. To download and install the latest version of Node.js, see [Node.js downloads](#).

However, if you prefer to use CommonJS syntax, please refer to [JavaScript ES6/CommonJS syntax](#).

Note

For information about the data types used in these examples, see [Supported data types and naming rules in Amazon DynamoDB](#).

Querying a table

Create a Node.js module with the file name `query.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. To access DynamoDB, create a DynamoDB client service object. Create a JSON object containing the parameters needed to query the table, which in this example includes the table name, the `ExpressionAttributeValues` needed by the query, a `KeyConditionExpression` that uses those values to define which items the query returns, and the names of attribute values to return for each item. Call the `QueryCommand` method of the DynamoDB service object.

```
import { DynamoDBClient, QueryCommand } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({});

export const main = async () => {
  const command = new QueryCommand({
    KeyConditionExpression: "Flavor = :flavor",
    // For more information about data types,
    // see https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
    // HowItWorks.NamingRulesDataTypes.html#HowItWorks.DataTypes and
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
    // Programming.LowLevelAPI.html#Programming.LowLevelAPI.DataTypeDescriptors
    ExpressionAttributeValues: {
      ":flavor": { S: "Key Lime" },
      ":searchKey": { S: "no coloring" },
    },
    FilterExpression: "contains (Description, :searchKey)",
    ProjectionExpression: "Flavor, CrustType, Description",
    TableName: "Pies",
  });

  const response = await client.send(command);
  response.Items.forEach(function (pie) {
    console.log(`${pie.Flavor.S} - ${pie.Description.S}\n`);
  });
};
```

```
});  
return response;  
};
```

To run the example, enter the following at the command prompt.

```
node query.js
```

This example code can be found [here on GitHub](#).

Scanning a table

Create a Node.js module with the file name `scan.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. To access DynamoDB, create a DynamoDB client service object. Create a JSON object containing the parameters needed to scan the table for items, which in this example includes the name of the table, the list of attribute values to return for each matching item, and an expression to filter the result set to find items containing a specified phrase. Call the `ScanCommand` method of the DynamoDB service object.

```
import { DynamoDBClient, ScanCommand } from "@aws-sdk/client-dynamodb";  
  
const client = new DynamoDBClient({});  
  
export const main = async () => {  
  const command = new ScanCommand({  
    FilterExpression: "CrustType = :crustType",  
    // For more information about data types,  
    // see https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/  
    // HowItWorks.NamingRulesDataTypes.html#HowItWorks.DataTypes and  
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/  
    // Programming.LowLevelAPI.html#Programming.LowLevelAPI.DataTypeDescriptors  
    ExpressionAttributeValues: {  
      ":crustType": { S: "Graham Cracker" },  
    },  
    ProjectionExpression: "Flavor, CrustType, Description",  
    TableName: "Pies",  
  });  
  
  const response = await client.send(command);  
  response.Items.forEach(function (pie) {  
    console.log(`${pie.Flavor.S} - ${pie.Description.S}\n`);  
  });  
};
```



```
    return response;
};
```

To run the example, enter the following at the command prompt.

```
node scan.js
```

This example code can be found [here on GitHub](#).

Using the DynamoDB Document Client



This Node.js code example shows:

- How to access a DynamoDB table using the DynamoDB utilities.

The Scenario

The DynamoDB Document Client simplifies working with items by abstracting the notion of attribute values. This abstraction annotates native JavaScript types supplied as input parameters, and converts annotated response data to native JavaScript types.

For more information about the DynamoDB Document Client, see [@aws-sdk/lib-dynamodb README](#) on GitHub. For more information about programming with Amazon DynamoDB, see [Programming Amazon DynamoDB with JavaScript](#) in the *Amazon DynamoDB Developer Guide*.

In this example, you use a series of Node.js modules to perform basic operations on a DynamoDB table using DynamoDB utilities. The code uses the SDK for JavaScript to query and scan tables using these methods of the DynamoDB Document Client class:

- [GetCommand](#)
- [PutCommand](#)
- [UpdateCommand](#)
- [QueryCommand](#)
- [DeleteCommand](#)

For more information on configuring the DynamoDB Document Client, see [@aws-sdk/lib-dynamodb](#).

Prerequisite Tasks

To set up and run this example, first complete these tasks:

- Set up the project environment to run these Node.js examples, and install the required AWS SDK for JavaScript and third-party modules. Follow the instructions on [GitHub](#).
- Create a shared configurations file with your user credentials. For more information about providing a shared credentials file, see [Shared config and credentials files](#) in the *AWS SDKs and Tools Reference Guide*.
- Create a DynamoDB table whose items you can access. For more information about creating a DynamoDB table using the SDK for JavaScript, see [Creating and using tables in DynamoDB](#). You can also use the [DynamoDB console](#) to create a table.

Important

These examples use ECMAScript6 (ES6). This requires Node.js version 13.x or higher. To download and install the latest version of Node.js, see [Node.js downloads](#).

However, if you prefer to use CommonJS syntax, please refer to [JavaScript ES6/CommonJS syntax](#).

Note

For information about the data types used in these examples, see [Supported data types and naming rules in Amazon DynamoDB](#).

Getting an Item from a Table

Create a Node.js module with the file name `get.js`. Be sure to configure the SDK as previously shown, including installing the required clients and packages. This includes `@aws-sdk/lib-dynamodb`, a library package that provides document client functionality to `@aws-sdk/client-dynamodb`. Next, set the configuration as shown below for marshalling and unmarshalling - as an optional second parameter - during creation of document client. Next, create the clients. Now

create a JSON object containing the parameters needed get an item from the table, which in this example includes the name of the table, the name of the hash key in that table, and the value of the hash key for the item you want to get. Call the `GetCommand` method of the `DynamoDB Document Client`.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, GetCommand } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new GetCommand({
    TableName: "AngryAnimals",
    Key: {
      CommonName: "Shoebill",
    },
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

To run the example, enter the following at the command prompt.

```
node get.js
```

This example code can be found [here on GitHub](#).

Putting an Item in a Table

Create a Node.js module with the file name `put.js`. Be sure to configure the SDK as previously shown, including installing the required clients and packages. This includes `@aws-sdk/lib-dynamodb`, a library package that provides document client functionality to `@aws-sdk/client-dynamodb`. Next, set the configuration as shown below for marshalling and unmarshalling - as an optional second parameter - during creation of document client. Next, create the clients. Create a JSON object containing the parameters needed to write an item to the table, which in this example includes the name of the table and a description of the item to add or update that includes the hashkey and value and names and values for attributes to set on the item. Call the `PutCommand` method of the `DynamoDB Document Client`.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { PutCommand, DynamoDBDocumentClient } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new PutCommand({
    TableName: "HappyAnimals",
    Item: {
      CommonName: "Shiba Inu",
    },
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

To run the example, enter the following at the command prompt.

```
node put.js
```

This example code can be found [here on GitHub](#).

Updating an Item in a Table

Create a Node.js module with the file name `update.js`. Be sure to configure the SDK as previously shown, including installing the required clients and packages. This includes `@aws-sdk/lib-dynamodb`, a library package that provides document client functionality to `@aws-sdk/client-dynamodb`. Next, set the configuration as shown below for marshalling and unmarshalling - as an optional second parameter - during creation of document client. Next, create the clients. Create a JSON object containing the parameters needed to write an item to the table, which in this example includes the name of the table, the key of the item to update, a set of `UpdateExpressions` that define the attributes of the item to update with tokens you assign values to in the `ExpressionAttributeValue` parameters. Call the `UpdateCommand` method of the `DynamoDB Document Client`.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, UpdateCommand } from "@aws-sdk/lib-dynamodb";
```

```
const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new UpdateCommand({
    TableName: "Dogs",
    Key: {
      Breed: "Labrador",
    },
    UpdateExpression: "set Color = :color",
    ExpressionAttributeValues: {
      ":color": "black",
    },
    ReturnValues: "ALL_NEW",
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

To run the example, enter the following at the command prompt.

```
node update.js
```

This example code can be found [here on GitHub](#).

Querying a Table

Create a Node.js module with the file name `query.js`. Be sure to configure the SDK as previously shown, including installing the required clients and packages. This includes `@aws-sdk/lib-dynamodb`, a library package that provides document client functionality to `@aws-sdk/client-dynamodb`. Create a JSON object containing the parameters needed to query the table, which in this example includes the table name, the `ExpressionAttributeValues` needed by the query, and a `KeyConditionExpression` that uses those values to define which items the query returns. Call the `QueryCommand` method of the DynamoDB Document Client.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { QueryCommand, DynamoDBDocumentClient } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);
```

```
export const main = async () => {
  const command = new QueryCommand({
    TableName: "CoffeeCrop",
    KeyConditionExpression:
      "OriginCountry = :originCountry AND RoastDate > :roastDate",
    ExpressionAttributeValues: {
      ":originCountry": "Ethiopia",
      ":roastDate": "2023-05-01",
    },
    ConsistentRead: true,
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

To run the example, enter the following at the command prompt.

```
node query.js
```

This example code can be found [here on GitHub](#).

Deleting an Item from a Table

Create a Node.js module with the file name `delete.js`. Be sure to configure the SDK as previously shown, including installing the required clients and packages. This includes `@aws-sdk/lib-dynamodb`, a library package that provides document client functionality to `@aws-sdk/client-dynamodb`. Next, set the configuration as shown below for marshalling and unmarshalling - as an optional second parameter - during creation of document client. Next, create the clients. To access DynamoDB, create a `DynamoDB` object. Create a JSON object containing the parameters needed to delete an item in the table, which in this example includes the name of the table and the name and value of the hashkey of the item you want to delete. Call the `DeleteCommand` method of the `DynamoDB Document Client`.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, DeleteCommand } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);
```

```
export const main = async () => {
  const command = new DeleteCommand({
    TableName: "Sodas",
    Key: {
      Flavor: "Cola",
    },
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

To run the example, enter the following at the command prompt.

```
node delete.js
```

This example code can be found [here on GitHub](#).

AWS Elemental MediaConvert examples

AWS Elemental MediaConvert is a file-based video transcoding service with broadcast-grade features. You can use it to create assets for broadcast and for video-on-demand (VOD) delivery across the internet. For more information, see the [AWS Elemental MediaConvert User Guide](#).

The JavaScript API for MediaConvert is exposed through the `MediaConvert` client class. For more information, see [Class: MediaConvert](#) in the API Reference.

Topics

- [Getting your region-specific endpoint for MediaConvert](#)
- [Creating and managing transcoding jobs in MediaConvert](#)
- [Using job templates in MediaConvert](#)

Getting your region-specific endpoint for MediaConvert



This Node.js code example shows:

- How to retrieve your region-specific endpoint from MediaConvert.

The scenario

In this example, you use a Node.js module to call MediaConvert and retrieve your region-specific endpoint. You can retrieve your endpoint URL from the service default endpoint and so do not yet need your region-specific endpoint. The code uses the SDK for JavaScript to retrieve this endpoint by using this method of the MediaConvert client class:

- [DescribeEndpointsCommand](#)

Prerequisite tasks

To set up and run this example, first complete these tasks:

- Set up the project environment to run these Node TypeScript examples, and install the required AWS SDK for JavaScript and third-party modules. Follow the instructions on [GitHub](#).
- Create a shared configurations file with your user credentials. For more information about providing a shared credentials file, see [Shared config and credentials files](#) in the *AWS SDKs and Tools Reference Guide*.
- Create an IAM role that gives MediaConvert access to your input files and the Amazon S3 buckets where your output files are stored. For details, see [Set up IAM permissions](#) in the *AWS Elemental MediaConvert User Guide*.

Important

This example uses ECMAScript6 (ES6). This requires Node.js version 13.x or higher. To download and install the latest version of Node.js, see [Node.js downloads](#).

However, if you prefer to use CommonJS syntax, please refer to [JavaScript ES6/CommonJS syntax](#).

Getting your endpoint URL

Create a `libs` directory, and create a Node.js module with the file name `emcClientGet.js`. Copy and paste the code below into it, which creates the MediaConvert client object. Replace **REGION** with your AWS Region.

```
import { MediaConvertClient } from "@aws-sdk/client-mediaconvert";
// Set the AWS Region.
const REGION = "REGION";
//Set the MediaConvert Service Object
const emcClientGet = new MediaConvertClient({ region: REGION });
export { emcClientGet };
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `emc_getendpoint.js`. Be sure to configure the SDK as previously shown, including installing the required clients and packages.

Create an object to pass the empty request parameters for the `DescribeEndpointsCommand` method of the MediaConvert client class. Then call the `DescribeEndpointsCommand` method.

```
// Import required AWS-SDK clients and commands for Node.js
import { DescribeEndpointsCommand } from "@aws-sdk/client-mediaconvert";
import { emcClientGet } from "../libs/emcClientGet.js";

//set the parameters.
const params = { MaxResults: 0 };

const run = async () => {
  try {
    // Create a new service object and set MediaConvert to customer endpoint
    const data = await emcClientGet.send(new DescribeEndpointsCommand(params));
    console.log("Your MediaConvert endpoint is ", data.Endpoints);
    return data;
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

To run the example, enter the following at the command prompt.

```
node emc_getendpoint.js
```

This example code can be found [here on GitHub](#).

Creating and managing transcoding jobs in MediaConvert



This Node.js code example shows:

- How to specify the region-specific endpoint to use with MediaConvert.
- How to create transcoding jobs in MediaConvert.
- How to cancel a transcoding job.
- How to retrieve the JSON for a completed transcoding job.
- How to retrieve a JSON array for up to 20 of the most recently created jobs.

The scenario

In this example, you use a Node.js module to call MediaConvert to create and manage transcoding jobs. The code uses the SDK for JavaScript to do this by using these methods of the MediaConvert client class:

- [CreateJobCommand](#)
- [CancelJobCommand](#)
- [GetJobCommand](#)
- [ListJobsCommand](#)

Prerequisite tasks

To set up and run this example, first complete these tasks:

- Set up the project environment to run these Node TypeScript examples, and install the required AWS SDK for JavaScript and third-party modules. Follow the instructions on [GitHub](#).

- Create a shared configurations file with your user credentials. For more information about providing a shared credentials file, see [Shared config and credentials files](#) in the *AWS SDKs and Tools Reference Guide*.
- Create and configure Amazon S3 buckets that provide storage for job input files and output files. For details, see [Create storage for files](#) in the *AWS Elemental MediaConvert User Guide*.
- Upload the input video to the Amazon S3 bucket you provisioned for input storage. For a list of supported input video codecs and containers, see [Supported input codecs and containers](#) in the *AWS Elemental MediaConvert User Guide*.
- Create an IAM role that gives MediaConvert access to your input files and the Amazon S3 buckets where your output files are stored. For details, see [Set up IAM permissions](#) in the *AWS Elemental MediaConvert User Guide*.

Important

This example uses ECMAScript6 (ES6). This requires Node.js version 13.x or higher. To download and install the latest version of Node.js, see [Node.js downloads](#).

However, if you prefer to use CommonJS syntax, please refer to [JavaScript ES6/CommonJS syntax](#).

Configuring the SDK

Configure the SDK as previously shown, including downloading the required clients and packages. Because MediaConvert uses custom endpoints for each account, you must also configure the MediaConvert client class to use your region-specific endpoint. To do this, set the endpoint parameter on `mediaconvert(endpoint)`.

```
// Import required AWS-SDK clients and commands for Node.js
import { CreateJobCommand } from "@aws-sdk/client-mediaconvert";
import { emcClient } from "./libs/emcClient.js";
```

Defining a simple transcoding job

Create a `libs` directory, and create a Node.js module with the file name `emcClient.js`. Copy and paste the code below into it, which creates the MediaConvert client object. Replace **REGION** with

your AWS Region. Replace *ENDPOINT* with your MediaConvert account endpoint, which you can on the **Account** page in the MediaConvert console.

```
import { MediaConvertClient } from "@aws-sdk/client-mediaconvert";
// Set the account end point.
const ENDPOINT = {
  endpoint: "https://ENDPOINT_UNIQUE_STRING.mediaconvert.REGION.amazonaws.com",
};
// Set the MediaConvert Service Object
const emcClient = new MediaConvertClient(ENDPOINT);
export { emcClient };
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `emc_createjob.js`. Be sure to configure the SDK as previously shown, including installing the required clients and packages. Create the JSON that defines the transcode job parameters.

These parameters are quite detailed. You can use the [AWS Elemental MediaConvert console](#) to generate the JSON job parameters by choosing your job settings in the console, and then choosing **Show job JSON** at the bottom of the **Job** section. This example shows the JSON for a simple job.

Note

Replace *JOB_QUEUE_ARN* with the MediaConvert job queue, *IAM_ROLE_ARN* with the Amazon Resource Name (ARN) of the IAM role, *OUTPUT_BUCKET_NAME* with the destination bucket name - for example, "s3://OUTPUT_BUCKET_NAME/", and *INPUT_BUCKET_AND_FILENAME* with the input bucket and filename - for example, "s3://INPUT_BUCKET/FILE_NAME".

```
const params = {
  Queue: "JOB_QUEUE_ARN", //JOB_QUEUE_ARN
  UserMetadata: {
    Customer: "Amazon",
  },
  Role: "IAM_ROLE_ARN", //IAM_ROLE_ARN
  Settings: {
    OutputGroups: [
      {
        Name: "File Group",
```

```
OutputGroupSettings: {
  Type: "FILE_GROUP_SETTINGS",
  FileGroupSettings: {
    Destination: "OUTPUT_BUCKET_NAME", //OUTPUT_BUCKET_NAME, e.g., "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",
    FileInput: "INPUT_BUCKET_AND_FILENAME", //INPUT_BUCKET_AND_FILENAME, e.g.,
"s3://BUCKET_NAME/FILE_NAME"
  },
],
TimecodeConfig: {
  Source: "EMBEDDED",
},
},
];

```

Creating a transcoding job

After creating the job parameters JSON, call the asynchronous `run` method to invoke a `MediaConvert` client service object, passing the parameters. The ID of the job created is returned in the response data.

```

const run = async () => {
  try {

```

```
const data = await emcClient.send(new CreateJobCommand(params));
console.log("Job created!", data);
return data;
} catch (err) {
  console.log("Error", err);
}
};
run();
```

To run the example, enter the following at the command prompt.

```
node emc_createjob.js
```

This full example code can be found [here on GitHub](#).

Canceling a transcoding job

Create a `libs` directory, and create a Node.js module with the file name `emcClient.js`. Copy and paste the code below into it, which creates the `MediaConvert` client object. Replace **REGION** with your AWS Region. Replace **ENDPOINT** with your MediaConvert account endpoint, which you can on the **Account** page in the MediaConvert console.

```
import { MediaConvertClient } from "@aws-sdk/client-mediaconvert";
// Set the account end point.
const ENDPOINT = {
  endpoint: "https://ENDPOINT_UNIQUE_STRING.mediaconvert.REGION.amazonaws.com",
};
// Set the MediaConvert Service Object
const emcClient = new MediaConvertClient(ENDPOINT);
export { emcClient };
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `emc_canceljob.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. Create the JSON that includes the ID of the job to cancel. Then call the `CancelJobCommand` method by creating a promise for invoking an `MediaConvert` client service object, passing the parameters. Handle the response in the promise callback.

Note

Replace *JOB_ID* with the ID of the job to cancel.

```
// Import required AWS-SDK clients and commands for Node.js
import { CancelJobCommand } from "@aws-sdk/client-mediaconvert";
import { emcClient } from "../libs/emcClient.js";

// Set the parameters
const params = { Id: "JOB_ID" }; //JOB_ID

const run = async () => {
  try {
    const data = await emcClient.send(new CancelJobCommand(params));
    console.log("Job " + params.Id + " is canceled");
    return data;
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

To run the example, enter the following at the command prompt.

```
node ec2_canceljob.js
```

This example code can be found [here on GitHub](#).

Listing recent transcoding jobs

Create a `libs` directory, and create a Node.js module with the file name `emcClient.js`. Copy and paste the code below into it, which creates the MediaConvert client object. Replace *REGION* with your AWS Region. Replace *ENDPOINT* with your MediaConvert account endpoint, which you can on the **Account** page in the MediaConvert console.

```
import { MediaConvertClient } from "@aws-sdk/client-mediaconvert";
// Set the account end point.
const ENDPOINT = {
  endpoint: "https://ENDPOINT_UNIQUE_STRING.mediaconvert.REGION.amazonaws.com",
};
```

```
// Set the MediaConvert Service Object
const emcClient = new MediaConvertClient(ENDPOINT);
export { emcClient };
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `emc_listjobs.js`. Be sure to configure the SDK as previously shown, including installing the required clients and packages.

Create the parameters JSON, including values to specify whether to sort the list in ASCENDING, or DESCENDING order, the Amazon Resource Name (ARN) of the job queue to check, and the status of jobs to include. Then call the `ListJobsCommand` method by creating a promise for invoking an `MediaConvert` client service object, passing the parameters.

Note

Replace *QUEUE_ARN* with the Amazon Resource Name (ARN) of the job queue to check, and *STATUS* with the status of the queue.

```
// Import required AWS-SDK clients and commands for Node.js
import { ListJobsCommand } from "@aws-sdk/client-mediaconvert";
import { emcClient } from "../libs/emcClient.js";

// Set the parameters
const params = {
  MaxResults: 10,
  Order: "ASCENDING",
  Queue: "QUEUE_ARN",
  Status: "SUBMITTED", // e.g., "SUBMITTED"
};

const run = async () => {
  try {
    const data = await emcClient.send(new ListJobsCommand(params));
    console.log("Success. Jobs: ", data.Jobs);
  } catch (err) {
    console.log("Error", err);
  }
};
```

```
run();
```

To run the example, enter the following at the command prompt.

```
node emc_listjobs.js
```

This example code can be found [here on GitHub](#).

Using job templates in MediaConvert



This Node.js code example shows:

- How to create AWS Elemental MediaConvert job templates.
- How to use a job template to create a transcoding job.
- How to list all your job templates.
- How to delete job templates.

The scenario

The JSON required to create a transcoding job in MediaConvert is detailed, containing a large number of settings. You can greatly simplify job creation by saving known-good settings in a job template that you can use to create subsequent jobs. In this example, you use a Node.js module to call MediaConvert to create, use, and manage job templates. The code uses the SDK for JavaScript to do this by using these methods of the MediaConvert client class:

- [CreateJobTemplateCommand](#)
- [CreateJobCommand](#)
- [DeleteJobTemplateCommand](#)
- [ListJobTemplatesCommand](#)

Prerequisite tasks

To set up and run this example, first complete these tasks:

- Set up the project environment to run these Node TypeScript examples, and install the required AWS SDK for JavaScript and third-party modules. Follow the instructions on [GitHub](#).
- Create a shared configurations file with your user credentials. For more information about providing a shared credentials file, see [Shared config and credentials files](#) in the *AWS SDKs and Tools Reference Guide*.
- Create an IAM role that gives MediaConvert access to your input files and the Amazon S3 buckets where your output files are stored. For details, see [Set up IAM permissions](#) in the *AWS Elemental MediaConvert User Guide*.

Important

These examples use ECMAScript6 (ES6). This requires Node.js version 13.x or higher. To download and install the latest version of Node.js, see [Node.js downloads](#).

However, if you prefer to use CommonJS syntax, please refer to [JavaScript ES6/CommonJS syntax](#).

Creating a job template

Create a `libs` directory, and create a Node.js module with the file name `emcClient.js`. Copy and paste the code below into it, which creates the MediaConvert client object. Replace **REGION** with your AWS Region. Replace **ENDPOINT** with your MediaConvert account endpoint, which you can on the **Account** page in the MediaConvert console.

```
import { MediaConvertClient } from "@aws-sdk/client-mediaconvert";
// Set the account end point.
const ENDPOINT = {
  endpoint: "https://ENDPOINT_UNIQUE_STRING.mediaconvert.REGION.amazonaws.com",
};
// Set the MediaConvert Service Object
const emcClient = new MediaConvertClient(ENDPOINT);
export { emcClient };
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `emc_create_jobtemplate.js`. Be sure to configure the SDK as previously shown, including installing the required clients and packages.

Specify the parameters JSON for template creation. You can use most of the JSON parameters from a previous successful job to specify the Settings values in the template. This example uses the job settings from [Creating and managing transcoding jobs in MediaConvert](#).

Call the `CreateJobTemplateCommand` method by creating a promise for invoking an `MediaConvert` client service object, passing the parameters.

Note

Replace `JOB_QUEUE_ARN` with the Amazon Resource Name (ARN) of the job queue to check, and `BUCKET_NAME` with the name of the destination Amazon S3 bucket - for example, `"s3://BUCKET_NAME/"`.

```
// Import required AWS-SDK clients and commands for Node.js
import { CreateJobTemplateCommand } from "@aws-sdk/client-mediaconvert";
import { emcClient } from "./libs/emcClient.js";

const params = {
  Category: "YouTube Jobs",
  Description: "Final production transcode",
  Name: "DemoTemplate",
  Queue: "JOB_QUEUE_ARN", //JOB_QUEUE_ARN
  Settings: {
    OutputGroups: [
      {
        Name: "File Group",
        OutputGroupSettings: {
          Type: "FILE_GROUP_SETTINGS",
          FileGroupSettings: {
            Destination: "BUCKET_NAME", // BUCKET_NAME e.g., "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",
  },
},
};

const run = async () => {
  try {
    // Create a promise on a MediaConvert object
    const data = await emcClient.send(new CreateJobTemplateCommand(params));
    console.log("Success!", data);
    return data;
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

To run the example, enter the following at the command prompt.

```
node emc_create_jobtemplate.js
```

This example code can be found [here on GitHub](#).

Creating a transcoding job from a job template

Create a `libs` directory, and create a Node.js module with the file name `emcClient.js`. Copy and paste the code below into it, which creates the MediaConvert client object. Replace **REGION** with your AWS Region. Replace **ENDPOINT** with your MediaConvert account endpoint, which you can on the **Account** page in the MediaConvert console.

```
import { MediaConvertClient } from "@aws-sdk/client-mediaconvert";
// Set the account end point.
```



```
const ENDPOINT = {
  endpoint: "https://ENDPOINT_UNIQUE_STRING.mediaconvert.REGION.amazonaws.com",
};
// Set the MediaConvert Service Object
const emcClient = new MediaConvertClient(ENDPOINT);
export { emcClient };
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `emc_template_createjob.js`. Be sure to configure the SDK as previously shown, including installing the required clients and packages.

Create the job creation parameters JSON, including the name of the job template to use, and the Settings to use that are specific to the job you're creating. Then call the `CreateJobsCommand` method by creating a promise for invoking an `MediaConvert` client service object, passing the parameters.

Note

Replace `JOB_QUEUE_ARN` with the Amazon Resource Name (ARN) of the job queue to check, `KEY_PAIR_NAME` with, `TEMPLATE_NAME` with, `ROLE_ARN` with the Amazon Resource Name (ARN) of the role, and `INPUT_BUCKET_AND_FILENAME` with the input bucket and filename - for example, `"s3://BUCKET_NAME/FILE_NAME"`.

```
// Import required AWS-SDK clients and commands for Node.js
import { CreateJobCommand } from "@aws-sdk/client-mediaconvert";
import { emcClient } from "../libs/emcClient.js";

const params = {
  Queue: "QUEUE_ARN", //QUEUE_ARN
  JobTemplate: "TEMPLATE_NAME", //TEMPLATE_NAME
  Role: "ROLE_ARN", //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: "INPUT_BUCKET_AND_FILENAME", //INPUT_BUCKET_AND_FILENAME, e.g.,
"s3://BUCKET_NAME/FILE_NAME"
},
],
},
};

const run = async () => {
    try {
        const data = await emcClient.send(new CreateJobCommand(params));
        console.log("Success! ", data);
        return data;
    } catch (err) {
        console.log("Error", err);
    }
};
run();
```

To run the example, enter the following at the command prompt.

```
node emc_template_createjob.js
```

This example code can be found [here on GitHub](#).

Listing your job templates

Create a `libs` directory, and create a Node.js module with the file name `emcClient.js`. Copy and paste the code below into it, which creates the MediaConvert client object. Replace **REGION** with your AWS Region. Replace **ENDPOINT** with your MediaConvert account endpoint, which you can on the **Account** page in the MediaConvert console.

```
import { MediaConvertClient } from "@aws-sdk/client-mediaconvert";
// Set the account end point.
const ENDPOINT = {
  endpoint: "https://ENDPOINT_UNIQUE_STRING.mediaconvert.REGION.amazonaws.com",
};
// Set the MediaConvert Service Object
const emcClient = new MediaConvertClient(ENDPOINT);
export { emcClient };
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `emc_listtemplates.js`. Be sure to configure the SDK as previously shown, including installing the required clients and packages.

Create an object to pass the request parameters for the `listTemplates` method of the `MediaConvert` client class. Include values to determine what templates to list (NAME, CREATION DATE, SYSTEM), how many to list, and their sort order. To call the `ListTemplatesCommand` method, create a promise for invoking an `MediaConvert` client service object, passing the parameters.

```
// Import required AWS-SDK clients and commands for Node.js
import { ListJobTemplatesCommand } from "@aws-sdk/client-mediaconvert";
import { emcClient } from "../libs/emcClient.js";

const params = {
  ListBy: "NAME",
  MaxResults: 10,
  Order: "ASCENDING",
};

const run = async () => {
  try {
    const data = await emcClient.send(new ListJobTemplatesCommand(params));
    console.log("Success ", data.JobTemplates);
    return data;
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

To run the example, enter the following at the command prompt.

```
node emc_listtemplates.js
```

This example code can be found [here on GitHub](#).

Deleting a job template

Create a `libs` directory, and create a Node.js module with the file name `emcClient.js`. Copy and paste the code below into it, which creates the MediaConvert client object. Replace **REGION** with your AWS Region. Replace **ENDPOINT** with your MediaConvert account endpoint, which you can on the **Account** page in the MediaConvert console.

```
import { MediaConvertClient } from "@aws-sdk/client-mediaconvert";
// Set the account end point.
const ENDPOINT = {
  endpoint: "https://ENDPOINT_UNIQUE_STRING.mediaconvert.REGION.amazonaws.com",
};
// Set the MediaConvert Service Object
const emcClient = new MediaConvertClient(ENDPOINT);
export { emcClient };
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `emc_deletetemplate.js`. Be sure to configure the SDK as previously shown, including installing the required clients and packages.

Create an object to pass the name of the job template you want to delete as parameters for the `DeleteJobTemplateCommand` method of the MediaConvert client class. To call the `DeleteJobTemplateCommand` method, create a promise for invoking an MediaConvert client service object, passing the parameters.

```
// Import required AWS-SDK clients and commands for Node.js
import { DeleteJobTemplateCommand } from "@aws-sdk/client-mediaconvert";
import { emcClient } from "./libs/emcClient.js";

// Set the parameters
const params = { Name: "test" }; //TEMPLATE_NAME

const run = async () => {
  try {
```

```
const data = await emcClient.send(new DeleteJobTemplateCommand(params));
console.log(
  "Success, template deleted! Request ID:",
  data.$metadata.requestId,
);
return data;
} catch (err) {
  console.log("Error", err);
}
};
run();
```

To run the example, enter the following at the command prompt.

```
node emc_deletetemplate.js
```

This example code can be found [here on GitHub](#).

AWS Lambda examples

AWS Lambda is a serverless compute service that lets you run code without provisioning or managing servers, creating workload-aware cluster scaling logic, maintaining event integrations, or managing runtimes.

The JavaScript API for AWS Lambda is exposed through the [LambdaService](#) client class.

Here are a list of examples that demonstrate how to create and use Lambda functions with the AWS SDK for JavaScript v3:

- [Invoking Lambda with API Gateway](#)
- [Creating scheduled events to execute AWS Lambda functions](#)

Amazon Lex examples

Amazon Lex is an AWS service for building conversational interfaces into applications using voice and text.

The JavaScript API for Amazon Lex is exposed through the [Lex Runtime Service](#) client class.

- [Building an Amazon Lex chatbot](#)

Amazon Polly examples



This Node.js code example shows:

- Upload audio recorded using Amazon Polly to Amazon S3

The scenario

In this example, a series of Node.js modules are used to automatically upload audio recorded using Amazon Polly to Amazon S3 using these methods of the Amazon S3 client class:

- [StartSpeechSynthesisTaskCommand](#)

Prerequisite tasks

To set up and run this example, you must first complete these tasks:

- Set up a project environment to run Node JavaScript examples by following the instructions on [GitHub](#).
- Create a shared configurations file with your user credentials. For more information about providing a shared credentials file, see [Shared config and credentials files](#) in the *AWS SDKs and Tools Reference Guide*.
- Create an AWS Identity and Access Management (IAM) Unauthenticated Amazon Cognito user role `polly:SynthesizeSpeech` permissions, and an Amazon Cognito identity pool with the IAM role attached to it. The [Create the AWS resources using the AWS CloudFormation](#) section below describes how to create these resources.

Note

This example uses Amazon Cognito, but if you are not using Amazon Cognito then your AWS user must have following IAM permissions policy

```
{
```

```
"Version": "2012-10-17",
"Statement": [
  {
    "Action": [
      "mobileanalytics:PutEvents",
      "cognito-sync:*"
    ],
    "Resource": "*",
    "Effect": "Allow"
  },
  {
    "Action": "polly:SynthesizeSpeech",
    "Resource": "*",
    "Effect": "Allow"
  }
]
```

Create the AWS resources using the AWS CloudFormation

AWS CloudFormation enables you to create and provision AWS infrastructure deployments predictably and repeatedly. For more information about AWS CloudFormation, see the [AWS CloudFormation User Guide](#).

To create the AWS CloudFormation stack:

1. Install and configure the AWS CLI following the instructions in the [AWS CLI User Guide](#).
2. Create a file named `setup.yaml` in the root directory of your project folder, and copy the content [here on GitHub](#) into it.

Note

The AWS CloudFormation template was generated using the AWS CDK available [here on GitHub](#). For more information about the AWS CDK, see the [AWS Cloud Development Kit \(AWS CDK\) Developer Guide](#).

3. Run the following command from the command line, replacing `STACK_NAME` with a unique name for the stack.

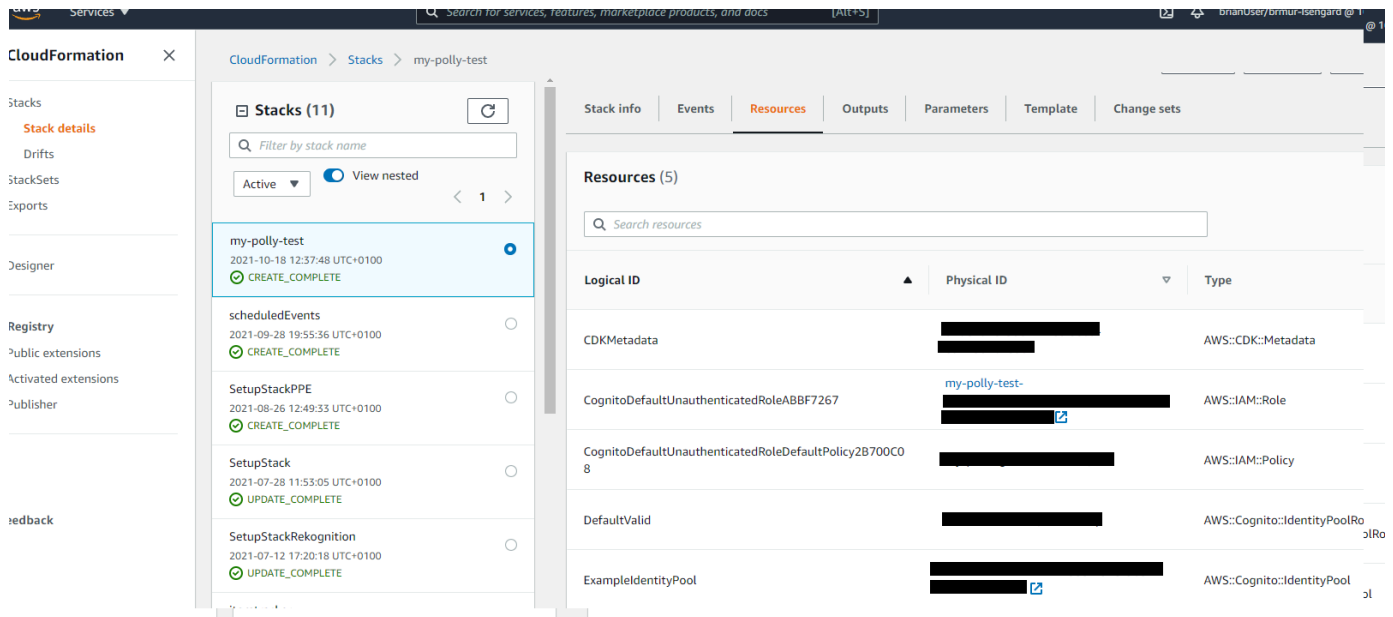
⚠ Important

The stack name must be unique within an AWS Region and AWS account. You can specify up to 128 characters, and numbers and hyphens are allowed.

```
aws cloudformation create-stack --stack-name STACK_NAME --template-body file://
setup.yaml --capabilities CAPABILITY_IAM
```

For more information on the `create-stack` command parameters, see the [AWS CLI Command Reference guide](#), and the [AWS CloudFormation User Guide](#).

4. Navigate to the AWS CloudFormation management console, choose **Stacks**, choose the stack name, and choose the **Resources** tab to view a list of the created resources.



Upload audio recorded using Amazon Polly to Amazon S3

Create a Node.js module with the file name `polly_synthesize_to_s3.js`. Make sure to configure the SDK as previously shown, including installing the required clients and packages. In the code, enter the **REGION**, and the **BUCKET_NAME**. To access Amazon Polly, create an Polly client service object. Replace **"IDENTITY_POOL_ID"** with the `IdentityPoolId` from the **Sample page** of the Amazon Cognito identity pool you created for this example. This is also passed to each client object.

Call the `StartSpeechSynthesisCommand` method of the Amazon Polly client service object to synthesize the voice message and upload it to the Amazon S3 bucket.

```
import { StartSpeechSynthesisTaskCommand } from "@aws-sdk/client-polly";
import { pollyClient } from "../libs/pollyClient.js";

// Create the parameters
var params = {
  OutputFormat: "mp3",
  OutputS3BucketName: "videoanalyzerbucket",
  Text: "Hello David, How are you?",
  TextType: "text",
  VoiceId: "Joanna",
  SampleRate: "22050",
};

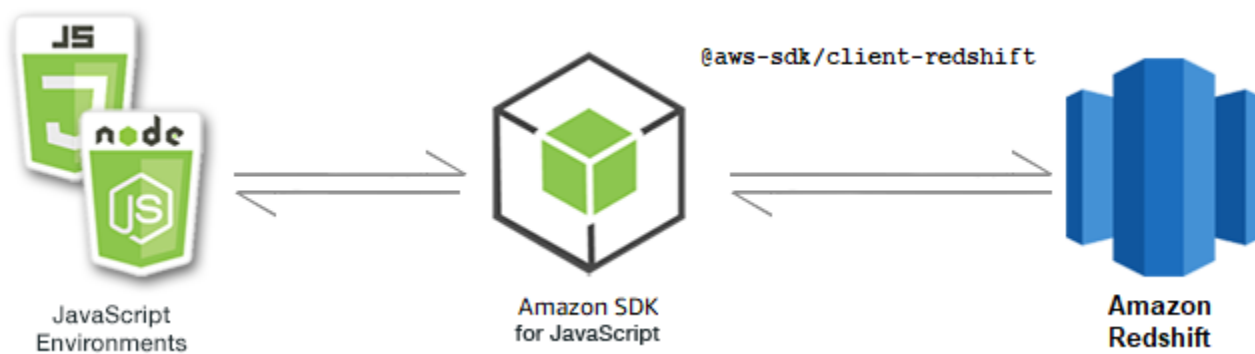
const run = async () => {
  try {
    await pollyClient.send(new StartSpeechSynthesisTaskCommand(params));
    console.log("Success, audio file added to " + params.OutputS3BucketName);
  } catch (err) {
    console.log("Error putting object", err);
  }
};

run();
```

This sample code can be found [here on GitHub](#).

Amazon Redshift examples

Amazon Redshift is a fully managed, petabyte-scale data warehouse service in the cloud. An Amazon Redshift data warehouse is a collection of computing resources called *nodes*, which are organized into a group called a *cluster*. Each cluster runs an Amazon Redshift engine and contains one or more databases.



The JavaScript API for Amazon Redshift is exposed through the [Amazon Redshift](#) client class.

Topics

- [Amazon Redshift examples](#)

Amazon Redshift examples

In this example, a series of Node.js modules are used to create, modify, describe the parameters of, and then delete Amazon Redshift clusters using the following methods of the Redshift client class:

- [CreateClusterCommand](#)
- [ModifyClusterCommand](#)
- [DescribeClustersCommand](#)
- [DeleteClusterCommand](#)

For more information about Amazon Redshift users, see the [Amazon Redshift getting started guide](#).

Prerequisite tasks

To set up and run this example, you must first complete these tasks:

- Set up the project environment to run these Node TypeScript examples, and install the required AWS SDK for JavaScript and third-party modules. Follow the instructions on [GitHub](#).
- Create a shared configurations file with your user credentials. For more information about providing a shared credentials file, see [Shared config and credentials files](#) in the *AWS SDKs and Tools Reference Guide*.

⚠ Important

These examples demonstrate how to import/export client service objects and command using ECMAScript6 (ES6).

- This requires Node.js version 13.x or higher. To download and install the latest version of Node.js, see [Node.js downloads](#).
- If you prefer to use CommonJS syntax, see [JavaScript ES6/CommonJS syntax](#)

Creating an Amazon Redshift cluster

This example demonstrates how to create an Amazon Redshift cluster using the AWS SDK for JavaScript. For more information, see [CreateCluster](#).

⚠ Important

The cluster that you are about to create is live (and not running in a sandbox). You incur the standard Amazon Redshift usage fees for the cluster until you delete it. If you delete the cluster in the same sitting as when you create it, the total charges are minimal.

Create a `libs` directory, and create a Node.js module with the file name `redshiftClient.js`. Copy and paste the code below into it, which creates the Amazon Redshift client object. Replace **REGION** with your AWS Region.

```
import { RedshiftClient } from "@aws-sdk/client-redshift";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create Redshift service object.
const redshiftClient = new RedshiftClient({ region: REGION });
export { redshiftClient };
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `redshift-create-cluster.js`. Make sure to configure the SDK as previously shown, including installing the required clients and packages. Create a `parameters` object, specifying the node type to be provisioned, and the master sign-in

credentials for the database instance automatically created in the cluster, and finally the cluster type.

Note

Replace *CLUSTER_NAME* with the name of the cluster. For *NODE_TYPE* specify the node type to be provisioned, such as 'dc2.large', for example. *MASTER_USERNAME* and *MASTER_USER_PASSWORD* are the sign-in credentials of the master user of your DB instance in the cluster. For *CLUSTER_TYPE*, enter the type of cluster. If you specify single-node, you do not require the `NumberOfNodes` parameter. The remaining parameters are optional.

```
// Import required AWS SDK clients and commands for Node.js
import { CreateClusterCommand } from "@aws-sdk/client-redshift";
import { redshiftClient } from "../libs/redshiftClient.js";

const params = {
  ClusterIdentifier: "CLUSTER_NAME", // Required
  NodeType: "NODE_TYPE", //Required
  MasterUsername: "MASTER_USER_NAME", // Required - must be lowercase
  MasterUserPassword: "MASTER_USER_PASSWORD", // Required - must contain at least one
  uppercase letter, and one number
  ClusterType: "CLUSTER_TYPE", // Required
  IAMRoleARN: "IAM_ROLE_ARN", // Optional - the ARN of an IAM role with permissions
  your cluster needs to access other AWS services on your behalf, such as Amazon S3.
  ClusterSubnetGroupName: "CLUSTER_SUBNET_GROUPNAME", //Optional - the name of a
  cluster subnet group to be associated with this cluster. Defaults to 'default' if not
  specified.
  DBName: "DATABASE_NAME", // Optional - defaults to 'dev' if not specified
  Port: "PORT_NUMBER", // Optional - defaults to '5439' if not specified
};

const run = async () => {
  try {
    const data = await redshiftClient.send(new CreateClusterCommand(params));
    console.log(
      "Cluster " + data.Cluster.ClusterIdentifier + " successfully created",
    );
    return data; // For unit tests.
  } catch (err) {
```

```
    console.log("Error", err);
  }
};
run();
```

To run the example, enter the following at the command prompt.

```
node redshift-create-cluster.js
```

This sample code can be found [here on GitHub](#).

Modifying a Amazon Redshift cluster

This example shows how to modify the master user password of an Amazon Redshift cluster using the AWS SDK for JavaScript. For more information about what other setting you can modify, see [ModifyCluster](#).

Create a `libs` directory, and create a Node.js module with the file name `redshiftClient.js`. Copy and paste the code below into it, which creates the Amazon Redshift client object. Replace **REGION** with your AWS Region.

```
import { RedshiftClient } from "@aws-sdk/client-redshift";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create Redshift service object.
const redshiftClient = new RedshiftClient({ region: REGION });
export { redshiftClient };
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `redshift-modify-cluster.js`. Make sure to configure the SDK as previously shown, including installing the required clients and packages. Specify the AWS Region, the name of the cluster you want to modify, and new master user password.

Note

Replace **CLUSTER_NAME** with the name of the cluster, and **MASTER_USER_PASSWORD** with the new master user password.

```
// Import required AWS SDK clients and commands for Node.js
import { ModifyClusterCommand } from "@aws-sdk/client-redshift";
import { redshiftClient } from "../libs/redshiftClient.js";

// Set the parameters
const params = {
  ClusterIdentifier: "CLUSTER_NAME",
  MasterUserPassword: "NEW_MASTER_USER_PASSWORD",
};

const run = async () => {
  try {
    const data = await redshiftClient.send(new ModifyClusterCommand(params));
    console.log("Success was modified.", data);
    return data; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

To run the example, enter the following at the command prompt.

```
node redshift-modify-cluster.js
```

This sample code can be found [here on GitHub](#).

Viewing details of a Amazon Redshift cluster

This example shows how to view the details of an Amazon Redshift cluster using the AWS SDK for JavaScript. For more information about optional, see [DescribeClusters](#).

Create a `libs` directory, and create a Node.js module with the file name `redshiftClient.js`. Copy and paste the code below into it, which creates the Amazon Redshift client object. Replace **REGION** with your AWS Region.

```
import { RedshiftClient } from "@aws-sdk/client-redshift";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create Redshift service object.
const redshiftClient = new RedshiftClient({ region: REGION });
export { redshiftClient };
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `redshift-describe-clusters.js`. Make sure to configure the SDK as previously shown, including installing the required clients and packages. Specify the AWS Region, the name of the cluster you want to modify, and new master user password.

Note

Replace *CLUSTER_NAME* with the name of the cluster.

```
// Import required AWS SDK clients and commands for Node.js
import { DescribeClustersCommand } from "@aws-sdk/client-redshift";
import { redshiftClient } from "../libs/redshiftClient.js";

const params = {
  ClusterIdentifier: "CLUSTER_NAME",
};

const run = async () => {
  try {
    const data = await redshiftClient.send(new DescribeClustersCommand(params));
    console.log("Success", data);
    return data; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};

run();
```

To run the example, enter the following at the command prompt.

```
node redshift-describe-clusters.js
```

This sample code can be found [here on GitHub](#).

Delete an Amazon Redshift cluster

This example shows how to view the details of an Amazon Redshift cluster using the AWS SDK for JavaScript. For more information about what other setting you can modify, see [DeleteCluster](#).

Create a `libs` directory, and create a Node.js module with the file name `redshiftClient.js`. Copy and paste the code below into it, which creates the Amazon Redshift client object. Replace **REGION** with your AWS Region.

```
import { RedshiftClient } from "@aws-sdk/client-redshift";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create Redshift service object.
const redshiftClient = new RedshiftClient({ region: REGION });
export { redshiftClient };
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file named `redshift-delete-clusters.js`. Make sure to configure the SDK as previously shown, including installing the required clients and packages. Specify the AWS Region, the name of the cluster you want to modify, and new master user password. The specify if you want to save a final snapshot of the cluster before deleting, and if so the ID of the snapshot.

Note

Replace **CLUSTER_NAME** with the name of the cluster. For the **SkipFinalClusterSnapshot**, specify whether to create a final snapshot of the cluster before deleting it. If you specify 'false', specify the id of the final cluster snapshot in **CLUSTER_SNAPSHOT_ID**. You can get this ID by clicking the link in the **Snapshots** column for the cluster on the **Clusters** dashboard, and scrolling down to the **Snapshots** pane. Note that the stem `rs:` is not part of the snapshot ID.

```
// Import required AWS SDK clients and commands for Node.js
import { DeleteClusterCommand } from "@aws-sdk/client-redshift";
import { redshiftClient } from "../libs/redshiftClient.js";

const params = {
  ClusterIdentifier: "CLUSTER_NAME",
  SkipFinalClusterSnapshot: false,
  FinalClusterSnapshotIdentifier: "CLUSTER_SNAPSHOT_ID",
};

const run = async () => {
```



```
try {
  const data = await redshiftClient.send(new DeleteClusterCommand(params));
  console.log("Success, cluster deleted. ", data);
  return data; // For unit tests.
} catch (err) {
  console.log("Error", err);
}
};
run();
```

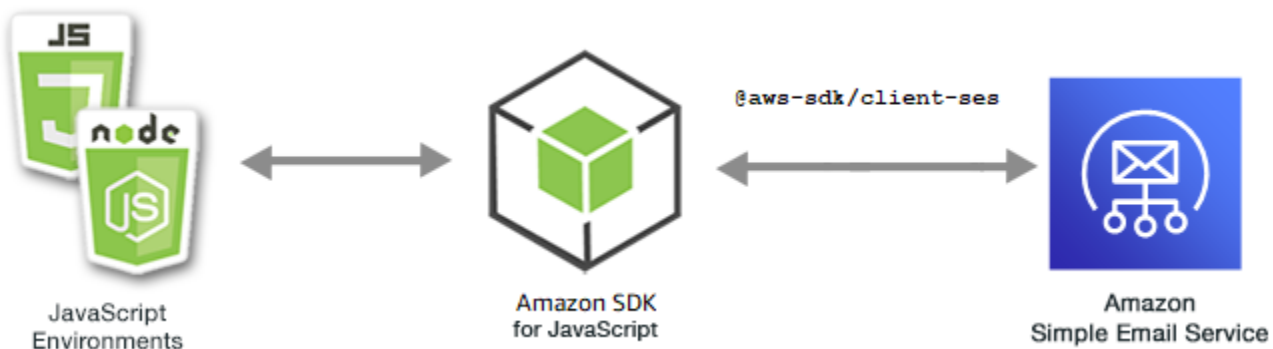
To run the example, enter the following at the command prompt.

```
node redshift-delete-cluster.js
```

This sample code can be found [here on GitHub](#).

Amazon Simple Email Service examples

Amazon Simple Email Service (Amazon SES) is a cloud-based email sending service designed to help digital marketers and application developers send marketing, notification, and transactional emails. It is a reliable, cost-effective service for businesses of all sizes that use email to keep in contact with their customers.



The JavaScript API for Amazon SES is exposed through the SES client class. For more information about using the Amazon SES client class, see [Class: SES](#) in the API Reference.

Topics

- [Managing Amazon SES identities](#)
- [Working with email templates in Amazon SES](#)
- [Sending email using Amazon SES](#)

Managing Amazon SES identities



This Node.js code example shows:

- How to verify email addresses and domains used with Amazon SES.
- How to assign an AWS Identity and Access Management (IAM) policy to your Amazon SES identities.
- How to list all Amazon SES identities for your AWS account.
- How to delete identities used with Amazon SES.

An Amazon SES *identity* is an email address or domain that Amazon SES uses to send email. Amazon SES requires you to verify your email identities, confirming that you own them and preventing others from using them.

For details on how to verify email addresses and domains in Amazon SES, see [Verifying email addresses and domains in Amazon SES](#) in the Amazon Simple Email Service Developer Guide. For information about sending authorization in Amazon SES, see [Overview of Amazon SES sending authorization](#).

The scenario

In this example, you use a series of Node.js modules to verify and manage Amazon SES identities. The Node.js modules use the SDK for JavaScript to verify email addresses and domains, using these methods of the SES client class:

- [ListIdentitiesCommand](#)
- [DeleteIdentityCommand](#)
- [VerifyEmailIdentityCommand](#)
- [VerifyDomainIdentityCommand](#)

Prerequisite tasks

To set up and run this example, you must first complete these tasks:

- Set up the project environment to run these Node TypeScript examples, and install the required AWS SDK for JavaScript and third-party modules. Follow the instructions on [GitHub](#).
- Create a shared configurations file with your user credentials. For more information about providing a shared credentials file, see [Shared config and credentials files](#) in the *AWS SDKs and Tools Reference Guide*.

Important

These examples demonstrate how to import/export client service objects and command using ECMAScript6 (ES6).

- This requires Node.js version 13.x or higher. To download and install the latest version of Node.js, see [Node.js downloads](#).
- If you prefer to use CommonJS syntax, see [JavaScript ES6/CommonJS syntax](#).

Listing your identities

In this example, use a Node.js module to list email addresses and domains to use with Amazon SES.

Create a `libs` directory, and create a Node.js module with the file name `sesClient.js`. Copy and paste the code below into it, which creates the Amazon SES client object. Replace *REGION* with your AWS Region.

```
import { SESClient } from "@aws-sdk/client-ses";
// Set the AWS Region.
const REGION = "us-east-1";
// Create SES service object.
const sesClient = new SESClient({ region: REGION });
export { sesClient };
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `ses_listidentities.js`. Configure the SDK as previously shown, including installing the required clients and packages.

Create an object to pass the `IdentityType` and other parameters for the `ListIdentitiesCommand` method of the SES client class. To call the `ListIdentitiesCommand` method, invoke an Amazon SES service object, passing the parameters object.

The data returned contains an array of domain identities as specified by the `IdentityType` parameter.

Note

Replace *IdentityType* with the identity type, which can be "EmailAddress" or "Domain".

```
import { ListIdentitiesCommand } from "@aws-sdk/client-ses";
import { sesClient } from "../libs/sesClient.js";

const createListIdentitiesCommand = () =>
  new ListIdentitiesCommand({ IdentityType: "EmailAddress", MaxItems: 10 });

const run = async () => {
  const listIdentitiesCommand = createListIdentitiesCommand();

  try {
    return await sesClient.send(listIdentitiesCommand);
  } catch (err) {
    console.log("Failed to list identities.", err);
    return err;
  }
};
```

To run the example, enter the following at the command prompt.

```
node ses_listidentities.js
```

This example code can be found [here on GitHub](#).

Verifying an email address identity

In this example, use a Node.js module to verify email senders to use with Amazon SES.

Create a `libs` directory, and create a Node.js module with the file name `sesClient.js`. Copy and paste the code below into it, which creates the Amazon SES client object. Replace *REGION* with your AWS Region.

```
import { SESClient } from "@aws-sdk/client-ses";
// Set the AWS Region.
```

```
const REGION = "us-east-1";
// Create SES service object.
const sesClient = new SESClient({ region: REGION });
export { sesClient };
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `ses_verifyemailidentity.js`. Configure the SDK as previously shown, including downloading the required clients and packages.

Create an object to pass the `EmailAddress` parameter for the `VerifyEmailIdentityCommand` method of the SES client class. To call the `VerifyEmailIdentityCommand` method, invoke an Amazon SES client service object, passing the parameters.

Note

Replace *EMAIL_ADDRESS* with the email address, such as `name@example.com`.

```
// Import required AWS SDK clients and commands for Node.js
import { VerifyEmailIdentityCommand } from "@aws-sdk/client-ses";
import { sesClient } from "../libs/sesClient.js";

const EMAIL_ADDRESS = "name@example.com";

const createVerifyEmailIdentityCommand = (emailAddress) => {
  return new VerifyEmailIdentityCommand({ EmailAddress: emailAddress });
};

const run = async () => {
  const verifyEmailIdentityCommand =
    createVerifyEmailIdentityCommand(EMAIL_ADDRESS);
  try {
    return await sesClient.send(verifyEmailIdentityCommand);
  } catch (err) {
    console.log("Failed to verify email identity.", err);
    return err;
  }
};
```

To run the example, enter the following at the command prompt. The domain is added to Amazon SES to be verified.

```
node ses_verifyemailidentity.js
```

This example code can be found [here on GitHub](#).

Verifying a Domain identity

In this example, use a Node.js module to verify email domains to use with Amazon SES.

Create a `libs` directory, and create a Node.js module with the file name `sesClient.js`. Copy and paste the code below into it, which creates the Amazon SES client object. Replace **REGION** with your AWS Region.

```
import { SESClient } from "@aws-sdk/client-ses";
// Set the AWS Region.
const REGION = "us-east-1";
// Create SES service object.
const sesClient = new SESClient({ region: REGION });
export { sesClient };
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `ses_verifydomainidentity.js`. Configure the SDK as previously shown, including installing the required clients and packages.

Create an object to pass the `Domain` parameter for the `VerifyDomainIdentityCommand` method of the SES client class. To call the `VerifyDomainIdentityCommand` method, invoke an Amazon SES client service object, passing the parameters object.

Note

This example imports and uses the required AWS Service V3 package clients, V3 commands, and uses the `send` method in an `async/await` pattern. You can create this example using V2 commands instead by making some minor changes. For details, see [Using v3 commands](#).

Note

Replace **DOMAIN_NAME** with the domain name.

```
import { VerifyDomainIdentityCommand } from "@aws-sdk/client-ses";
import {
  getUniqueName,
  postfix,
} from "@aws-doc-sdk-examples/lib/utils/util-string.js";
import { sesClient } from "../libs/sesClient.js";

/**
 * You must have access to the domain's DNS settings to complete the
 * domain verification process.
 */
const DOMAIN_NAME = postfix(getUniqueName("Domain"), ".example.com");

const createVerifyDomainIdentityCommand = () => {
  return new VerifyDomainIdentityCommand({ Domain: DOMAIN_NAME });
};

const run = async () => {
  const VerifyDomainIdentityCommand = createVerifyDomainIdentityCommand();

  try {
    return await sesClient.send(VerifyDomainIdentityCommand);
  } catch (err) {
    console.log("Failed to verify domain.", err);
    return err;
  }
};
```

To run the example, enter the following at the command prompt. The domain is added to Amazon SES to be verified.

```
node ses_verifydomainidentity.js
```

This example code can be found [here on GitHub](#).

Deleting identities

In this example, use a Node.js module to delete email addresses or domains used with Amazon SES.

Create a `libs` directory, and create a Node.js module with the file name `sesClient.js`. Copy and paste the code below into it, which creates the Amazon SES client object. Replace *REGION* with your AWS Region.

```
import { SESClient } from "@aws-sdk/client-ses";
// Set the AWS Region.
const REGION = "us-east-1";
// Create SES service object.
const sesClient = new SESClient({ region: REGION });
export { sesClient };
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `ses_deleteidentity.js`. Configure the SDK as previously shown, including installing the required clients and packages.

Create an object to pass the `Identity` parameter for the `DeleteIdentityCommand` method of the SES client class. To call the `DeleteIdentityCommand` method, create a request for invoking an Amazon SES client service object, passing the parameters.

Note

This example imports and uses the required AWS Service V3 package clients, V3 commands, and uses the `send` method in an `async/await` pattern. You can create this example using V2 commands instead by making some minor changes. For details, see [Using v3 commands](#).

Note

Replace `IDENTITY_EMAIL` with the email of the identity to be deleted.

```
import { DeleteIdentityCommand } from "@aws-sdk/client-ses";
import { sesClient } from "../libs/sesClient.js";

const IDENTITY_EMAIL = "fake@example.com";

const createDeleteIdentityCommand = (identityName) => {
  return new DeleteIdentityCommand({
    Identity: identityName,
  });
};
```



```
const run = async () => {
  const deleteIdentityCommand = createDeleteIdentityCommand(IDENTITY_EMAIL);

  try {
    return await sesClient.send(deleteIdentityCommand);
  } catch (err) {
    console.log("Failed to delete identity.", err);
    return err;
  }
};
```

To run the example, enter the following at the command prompt.

```
node ses_deleteidentity.js
```

This example code can be found [here on GitHub](#).

Working with email templates in Amazon SES



This Node.js code example shows:

- How to get a list of all of your email templates.
- How to retrieve and update email templates.
- How to create and delete email templates.

Amazon SES enables you to send personalized email messages using email templates. For details on how to create and use email templates in Amazon SES, see [Sending personalized email using the Amazon SES API](#) in the Amazon Simple Email Service Developer Guide.

The scenario

In this example, you use a series of Node.js modules to work with email templates. The Node.js modules use the SDK for JavaScript to create and use email templates using these methods of the SES client class:

- [ListTemplatesCommand](#)
- [CreateTemplateCommand](#)
- [GetTemplateCommand](#)
- [DeleteTemplateCommand](#)
- [UpdateTemplateCommand](#)

Prerequisite tasks

To set up and run this example, you must first complete these tasks:

- Set up the project environment to run these Node TypeScript examples, and install the required AWS SDK for JavaScript and third-party modules. Follow the instructions on [GitHub](#).
- Create a shared configurations file with your user credentials. For more information about providing a shared credentials file, see [Shared config and credentials files](#) in the *AWS SDKs and Tools Reference Guide*.

Important

These examples demonstrate how to import/export client service objects and command using ECMAScript6 (ES6).

- This requires Node.js version 13.x or higher. To download and install the latest version of Node.js, see [Node.js downloads](#).
- If you prefer to use CommonJS syntax, see [JavaScript ES6/CommonJS syntax](#).

Listing your email templates

In this example, use a Node.js module to create an email template to use with Amazon SES.

Create a `libs` directory, and create a Node.js module with the file name `sesClient.js`. Copy and paste the code below into it, which creates the Amazon SES client object. Replace **REGION** with your AWS Region.

```
import { SESClient } from "@aws-sdk/client-ses";  
// Set the AWS Region.
```

```
const REGION = "us-east-1";
// Create SES service object.
const sesClient = new SESClient({ region: REGION });
export { sesClient };
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `ses_listtemplates.js`. Configure the SDK as previously shown, including installing the required clients and packages.

Create an object to pass the parameters for the `ListTemplatesCommand` method of the SES client class. To call the `ListTemplatesCommand` method, invoke an Amazon SES client service object, passing the parameters.

Note

This example imports and uses the required AWS Service V3 package clients, V3 commands, and uses the `send` method in an `async/await` pattern. You can create this example using V2 commands instead by making some minor changes. For details, see [Using v3 commands](#).

```
import { ListTemplatesCommand } from "@aws-sdk/client-ses";
import { sesClient } from "../libs/sesClient.js";

const createListTemplatesCommand = (maxItems) =>
  new ListTemplatesCommand({ MaxItems: maxItems });

const run = async () => {
  const listTemplatesCommand = createListTemplatesCommand(10);

  try {
    return await sesClient.send(listTemplatesCommand);
  } catch (err) {
    console.log("Failed to list templates.", err);
    return err;
  }
};
```

To run the example, enter the following at the command prompt. Amazon SES returns the list of templates.

```
node ses_listtemplates.js
```

This example code can be found [here on GitHub](#).

Getting an email template

In this example, use a Node.js module to get an email template to use with Amazon SES.

Create a `libs` directory, and create a Node.js module with the file name `sesClient.js`. Copy and paste the code below into it, which creates the Amazon SES client object. Replace **REGION** with your AWS Region.

```
import { SESClient } from "@aws-sdk/client-ses";
// Set the AWS Region.
const REGION = "us-east-1";
// Create SES service object.
const sesClient = new SESClient({ region: REGION });
export { sesClient };
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `ses_gettemplate.js`. Configure the SDK as previously shown, including installing the required clients and packages.

Create an object to pass the `TemplateName` parameter for the `GetTemplateCommand` method of the SES client class. To call the `GetTemplateCommand` method, invoke an Amazon SES client service object, passing the parameters.

Note

This example imports and uses the required AWS Service V3 package clients, V3 commands, and uses the `send` method in an `async/await` pattern. You can create this example using V2 commands instead by making some minor changes. For details, see [Using v3 commands](#).

Note

Replace **TEMPLATE_NAME** with the name of the template to return.

```
import { GetTemplateCommand } from "@aws-sdk/client-ses";
import { getUniqueName } from "@aws-doc-sdk-examples/lib/utils/util-string.js";
import { sesClient } from "../libs/sesClient.js";

const TEMPLATE_NAME = getUniqueName("TemplateName");

const createGetTemplateCommand = (templateName) =>
  new GetTemplateCommand({ TemplateName: templateName });

const run = async () => {
  const getTemplateCommand = createGetTemplateCommand(TEMPLATE_NAME);

  try {
    return await sesClient.send(getTemplateCommand);
  } catch (caught) {
    if (caught instanceof Error && caught.name === "MessageRejected") {
      /** @type { import('@aws-sdk/client-ses').MessageRejected } */
      const messageRejectedError = caught;
      return messageRejectedError;
    }
    throw caught;
  }
};
```

To run the example, enter the following at the command prompt. Amazon SES returns the template details.

```
node ses_gettemplate.js
```

This example code can be found [here on GitHub](#).

Creating an email template

In this example, use a Node.js module to create an email template to use with Amazon SES.

Create a `libs` directory, and create a Node.js module with the file name `sesClient.js`. Copy and paste the code below into it, which creates the Amazon SES client object. Replace *REGION* with your AWS Region.

```
import { SESClient } from "@aws-sdk/client-ses";
// Set the AWS Region.
```

```
const REGION = "us-east-1";
// Create SES service object.
const sesClient = new SESClient({ region: REGION });
export { sesClient };
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `ses_createtemplate.js`. Configure the SDK as previously shown, including installing the required clients and packages.

Create an object to pass the parameters for the `CreateTemplateCommand` method of the SES client class, including `TemplateName`, `HtmlPart`, `SubjectPart`, and `TextPart`. To call the `CreateTemplateCommand` method, invoke an Amazon SES client service object, passing the parameters.

Note

This example imports and uses the required AWS Service V3 package clients, V3 commands, and uses the `send` method in an `async/await` pattern. You can create this example using V2 commands instead by making some minor changes. For details, see [Using v3 commands](#).

Note

Replace *TEMPLATE_NAME* with a name for the new template, *HtmlPart* with the HTML tagged content of email, and *SubjectPart* with the subject of the email.

```
import { CreateTemplateCommand } from "@aws-sdk/client-ses";
import { sesClient } from "../libs/sesClient.js";
import { getUniqueName } from "@aws-doc-sdk-examples/lib/utils/util-string.js";

const TEMPLATE_NAME = getUniqueName("TestTemplateName");

const createCreateTemplateCommand = () => {
  return new CreateTemplateCommand({
    /**
     * The template feature in Amazon SES is based on the Handlebars template system.
```

```
*/
Template: {
  /**
   * The name of an existing template in Amazon SES.
   */
  TemplateName: TEMPLATE_NAME,
  HtmlPart: `
    <h1>Hello, {{contact.firstName}}!</h1>
    <p>
      Did you know Amazon has a mascot named Peccy?
    </p>
  `,
  SubjectPart: "Amazon Tip",
},
});
};

const run = async () => {
  const createTemplateCommand = createCreateTemplateCommand();

  try {
    return await sesClient.send(createTemplateCommand);
  } catch (err) {
    console.log("Failed to create template.", err);
    return err;
  }
};
```

To run the example, enter the following at the command prompt. The template is added to Amazon SES.

```
node ses_createtemplate.js
```

This example code can be found [here on GitHub](#).

Updating an email template

In this example, use a Node.js module to create an email template to use with Amazon SES.

Create a `libs` directory, and create a Node.js module with the file name `sesClient.js`. Copy and paste the code below into it, which creates the Amazon SES client object. Replace *REGION* with your AWS Region.

```
import { SESClient } from "@aws-sdk/client-ses";
// Set the AWS Region.
const REGION = "us-east-1";
// Create SES service object.
const sesClient = new SESClient({ region: REGION });
export { sesClient };
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `ses_updateTemplate.js`. Configure the SDK as previously shown, including installing the required clients and packages.

Create an object to pass the `Template` parameter values you want to update in the template, with the required `TemplateName` parameter passed to the `UpdateTemplateCommand` method of the SES client class. To call the `UpdateTemplateCommand` method, invoke an Amazon SES service object, passing the parameters.

Note

This example imports and uses the required AWS Service V3 package clients, V3 commands, and uses the `send` method in an `async/await` pattern. You can create this example using V2 commands instead by making some minor changes. For details, see [Using v3 commands](#).

Note

Replace `TEMPLATE_NAME` with a name of the template and `HTML_PART` with the HTML tagged content of email.

```
import { UpdateTemplateCommand } from "@aws-sdk/client-ses";
import { getUniqueName } from "@aws-doc-sdk-examples/lib/utils/util-string.js";
import { sesClient } from "../libs/sesClient.js";

const TEMPLATE_NAME = getUniqueName("TemplateName");
const HTML_PART = "<h1>Hello, World!</h1>";

const createUpdateTemplateCommand = () => {
```



```
return new UpdateTemplateCommand({
  Template: {
    TemplateName: TEMPLATE_NAME,
    HtmlPart: HTML_PART,
    SubjectPart: "Example",
    TextPart: "Updated template text.",
  },
});
};

const run = async () => {
  const updateTemplateCommand = createUpdateTemplateCommand();

  try {
    return await sesClient.send(updateTemplateCommand);
  } catch (err) {
    console.log("Failed to update template.", err);
    return err;
  }
};
```

To run the example, enter the following at the command prompt. Amazon SES returns the template details.

```
node ses_updatetemplate.js
```

This example code can be found [here on GitHub](#).

Deleting an email template

In this example, use a Node.js module to create an email template to use with Amazon SES.

Create a `libs` directory, and create a Node.js module with the file name `sesClient.js`. Copy and paste the code below into it, which creates the Amazon SES client object. Replace *REGION* with your AWS Region.

```
import { SESClient } from "@aws-sdk/client-ses";
// Set the AWS Region.
const REGION = "us-east-1";
// Create SES service object.
const sesClient = new SESClient({ region: REGION });
export { sesClient };
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `ses_deleteTemplate.js`. Configure the SDK as previously shown, including installing the required clients and packages.

Create an object to pass the `requiredTemplateName` parameter to the `DeleteTemplateCommand` method of the SES client class. To call the `DeleteTemplateCommand` method, invoke an Amazon SES service object, passing the parameters.

Note

This example imports and uses the required AWS Service V3 package clients, V3 commands, and uses the `send` method in an `async/await` pattern. You can create this example using V2 commands instead by making some minor changes. For details, see [Using v3 commands](#).

Note

Replace `TEMPLATE_NAME` with the name of the template to be deleted.

```
import { DeleteTemplateCommand } from "@aws-sdk/client-ses";
import { getUniqueName } from "@aws-doc-sdk-examples/lib/utils/util-string.js";
import { sesClient } from "../libs/sesClient.js";

const TEMPLATE_NAME = getUniqueName("TemplateName");

const createDeleteTemplateCommand = (templateName) =>
  new DeleteTemplateCommand({ TemplateName: templateName });

const run = async () => {
  const deleteTemplateCommand = createDeleteTemplateCommand(TEMPLATE_NAME);

  try {
    return await sesClient.send(deleteTemplateCommand);
  } catch (err) {
    console.log("Failed to delete template.", err);
    return err;
  }
}
```

```
};
```

To run the example, enter the following at the command prompt. Amazon SES returns the template details.

```
node ses_deletetemplate.js
```

This example code can be found [here on GitHub](#).

Sending email using Amazon SES



This Node.js code example shows:

- Send a text or HTML email.
- Send emails based on an email template.
- Send bulk emails based on an email template.

The Amazon SES API provides two different ways for you to send an email, depending on how much control you want over the composition of the email message: formatted and raw. For details, see [Sending formatted email using the Amazon SES API](#) and [Sending raw email using the Amazon SES API](#).

The scenario

In this example, you use a series of Node.js modules to send email in a variety of ways. The Node.js modules use the SDK for JavaScript to create and use email templates using these methods of the SES client class:

- [SendEmailCommand](#)
- [SendTemplatedEmailCommand](#)
- [SendBulkTemplatedEmailCommand](#)

Prerequisite tasks

To set up and run this example, you must first complete these tasks:

- Set up the project environment to run these Node TypeScript examples, and install the required AWS SDK for JavaScript and third-party modules. Follow the instructions on [GitHub](#).
- Create a shared configurations file with your user credentials. For more information about providing a shared credentials file, see [Shared config and credentials files](#) in the *AWS SDKs and Tools Reference Guide*.

Important

These examples demonstrate how to import/export client service objects and command using ECMAScript6 (ES6).

- This requires Node.js version 13.x or higher. To download and install the latest version of Node.js, see [Node.js downloads](#).
- If you prefer to use CommonJS syntax, see [JavaScript ES6/CommonJS syntax](#).

Email message sending requirements

Amazon SES composes an email message and immediately queues it for sending. To send email using the `SendEmailCommand` method, your message must meet the following requirements:

- You must send the message from a verified email address or domain. If you attempt to send email using a non-verified address or domain, the operation results in an "Email address not verified" error.
- If your account is still in the Amazon SES sandbox, you can only send to verified addresses or domains, or to email addresses associated with the Amazon SES Mailbox Simulator. For more information, see [Verifying email addresses and domains](#) in the Amazon Simple Email Service Developer Guide.
- The total size of the message, including attachments, must be smaller than 10 MB.
- The message must include at least one recipient email address. The recipient address can be a To: address, a CC: address, or a BCC: address. If a recipient email address is not valid (that is, it is

not in the format `UserName@[SubDomain.]Domain.TopLevelDomain`), the entire message is rejected, even if the message contains other recipients that are valid.

- The message cannot include more than 50 recipients across the To:, CC: and BCC: fields. If you need to send an email message to a larger audience, you can divide your recipient list into groups of 50 or fewer, and then call the `sendEmail` method several times to send the message to each group.

Sending an email

In this example, use a Node.js module to send email with Amazon SES.

Create a `libs` directory, and create a Node.js module with the file name `sesClient.js`. Copy and paste the code below into it, which creates the Amazon SES client object. Replace **REGION** with your AWS Region.

```
import { SESClient } from "@aws-sdk/client-ses";
// Set the AWS Region.
const REGION = "us-east-1";
// Create SES service object.
const sesClient = new SESClient({ region: REGION });
export { sesClient };
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `ses_sendemail.js`. Configure the SDK as previously shown, including installing the required clients and packages.

Create an object to pass the parameter values that define the email to be sent, including sender and receiver addresses, subject, and email body in plain text and HTML formats, to the `SendEmailCommand` method of the SES client class. To call the `SendEmailCommand` method, invoke an Amazon SES service object, passing the parameters.

Note

This example imports and uses the required AWS Service V3 package clients, V3 commands, and uses the `send` method in an `async/await` pattern. You can create this example using V2 commands instead by making some minor changes. For details, see [Using v3 commands](#).

Note

Replace *toAddress* with the address to send the email to, and *fromAddress* with the email address to send the email from.

```
import { SendEmailCommand } from "@aws-sdk/client-ses";
import { sesClient } from "../libs/sesClient.js";

const createSendEmailCommand = (toAddress, fromAddress) => {
  return new SendEmailCommand({
    Destination: {
      /* required */
      CcAddresses: [
        /* more items */
      ],
      ToAddresses: [
        toAddress,
        /* more To-email addresses */
      ],
    },
    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: "EMAIL_SUBJECT",
      },
    },
    Source: fromAddress,
    ReplyToAddresses: [
      /* more items */
    ]
  });
}
```

```
    ],
  });
};

const run = async () => {
  const sendEmailCommand = createSendEmailCommand(
    "recipient@example.com",
    "sender@example.com",
  );

  try {
    return await sesClient.send(sendEmailCommand);
  } catch (caught) {
    if (caught instanceof Error && caught.name === "MessageRejected") {
      /** @type { import('@aws-sdk/client-ses').MessageRejected } */
      const messageRejectedError = caught;
      return messageRejectedError;
    }
    throw caught;
  }
};
```

To run the example, enter the following at the command prompt. The email is queued for sending by Amazon SES.

```
node ses_sendemail.js
```

This example code can be found [found here on GitHub](#).

Sending an email using a template

In this example, use a Node.js module to send email with Amazon SES. Create a Node.js module with the file name `ses_sendtemplatedemail.js`. Configure the SDK as previously shown, including installing the required clients and packages.

Create an object to pass the parameter values that define the email to be sent, including sender and receiver addresses, subject, email body in plain text and HTML formats, to the `SendTemplatedEmailCommand` method of the SES client class. To call the `SendTemplatedEmailCommand` method, invoke an Amazon SES client service object, passing the parameters.

Note

This example imports and uses the required AWS Service V3 package clients, V3 commands, and uses the send method in an async/await pattern. You can create this example using V2 commands instead by making some minor changes. For details, see [Using v3 commands](#).

Note

Replace *REGION* with your AWS Region, *USER* with the name and email address to send the email to, *VERIFIED_EMAIL* with the email address to send the email from, and *TEMPLATE_NAME* with the name of the template.

```
import { SendTemplatedEmailCommand } from "@aws-sdk/client-ses";
import {
  getUniqueName,
  postfix,
} from "@aws-doc-sdk-examples/lib/utils/util-string.js";
import { sesClient } from "../libs/sesClient.js";

/**
 * Replace this with the name of an existing template.
 */
const TEMPLATE_NAME = getUniqueName("ReminderTemplate");

/**
 * Replace these with existing verified emails.
 */
const VERIFIED_EMAIL = postfix(getUniqueName("Bilbo"), "@example.com");

const USER = { firstName: "Bilbo", emailAddress: VERIFIED_EMAIL };

/**
 *
 * @param { { emailAddress: string, firstName: string } } user
 * @param { string } templateName - The name of an existing template in Amazon SES.
 * @returns { SendTemplatedEmailCommand }
 */
const createReminderEmailCommand = (user, templateName) => {
```



```
return new SendTemplatedEmailCommand({
  /**
   * Here's an example of how a template would be replaced with user data:
   * Template: <h1>Hello {{contact.firstName}},</h1><p>Don't forget about the party
  gifts!</p>
   * Destination: <h1>Hello Bilbo,</h1><p>Don't forget about the party gifts!</p>
   */
  Destination: { ToAddresses: [user.emailAddress] },
  TemplateData: JSON.stringify({ contact: { firstName: user.firstName } }),
  Source: VERIFIED_EMAIL,
  Template: templateName,
});
};

const run = async () => {
  const sendReminderEmailCommand = createReminderEmailCommand(
    USER,
    TEMPLATE_NAME,
  );
  try {
    return await sesClient.send(sendReminderEmailCommand);
  } catch (caught) {
    if (caught instanceof Error && caught.name === "MessageRejected") {
      /** @type { import('@aws-sdk/client-ses').MessageRejected} */
      const messageRejectedError = caught;
      return messageRejectedError;
    }
    throw caught;
  }
};
```

To run the example, enter the following at the command prompt. The email is queued for sending by Amazon SES.

```
node ses_sendtemplatedemail.js
```

This example code can be found [here on GitHub](#).

Sending bulk email using a template

In this example, use a Node.js module to send email with Amazon SES.

Create a `libs` directory, and create a Node.js module with the file name `sesClient.js`. Copy and paste the code below into it, which creates the Amazon SES client object. Replace *REGION* with your AWS Region.

```
import { SESClient } from "@aws-sdk/client-ses";
// Set the AWS Region.
const REGION = "us-east-1";
// Create SES service object.
const sesClient = new SESClient({ region: REGION });
export { sesClient };
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `ses_sendbulktemplatedemail.js`. Configure the SDK as previously shown, including installing the required clients and packages.

Create an object to pass the parameter values that define the email to be sent, including sender and receiver addresses, subject, and email body in plain text and HTML formats, to the `SendBulkTemplatedEmailCommand` method of the SES client class. To call the `SendBulkTemplatedEmailCommand` method, invoke an Amazon SES service object, passing the parameters.

Note

This example imports and uses the required AWS Service V3 package clients, V3 commands, and uses the `send` method in an `async/await` pattern. You can create this example using V2 commands instead by making some minor changes. For details, see [Using v3 commands](#).

Note

Replace *USERS* with the names and email addresses to send the email to, *VERIFIED_EMAIL_1* with the email address to send the email from, and *TEMPLATE_NAME* with the name of the template.

```
import { SendBulkTemplatedEmailCommand } from "@aws-sdk/client-ses";
import {
```

```
    getUniqueName,
    postfix,
  } from "@aws-doc-sdk-examples/lib/utils/util-string.js";
import { sesClient } from "../libs/sesClient.js";

/**
 * Replace this with the name of an existing template.
 */
const TEMPLATE_NAME = getUniqueName("ReminderTemplate");

/**
 * Replace these with existing verified emails.
 */
const VERIFIED_EMAIL_1 = postfix(getUniqueName("Bilbo"), "@example.com");
const VERIFIED_EMAIL_2 = postfix(getUniqueName("Frodo"), "@example.com");

const USERS = [
  { firstName: "Bilbo", emailAddress: VERIFIED_EMAIL_1 },
  { firstName: "Frodo", emailAddress: VERIFIED_EMAIL_2 },
];

/**
 *
 * @param { { emailAddress: string, firstName: string }[] } users
 * @param { string } templateName the name of an existing template in SES
 * @returns { SendBulkTemplatedEmailCommand }
 */
const createBulkReminderEmailCommand = (users, templateName) => {
  return new SendBulkTemplatedEmailCommand({
    /**
     * Each 'Destination' uses a corresponding set of replacement data. We can map each
     user
     * to a 'Destination' and provide user specific replacement data to create
     personalized emails.
     *
     * Here's an example of how a template would be replaced with user data:
     * Template: <h1>Hello {{name}},</h1><p>Don't forget about the party gifts!</p>
     * Destination 1: <h1>Hello Bilbo,</h1><p>Don't forget about the party gifts!</p>
     * Destination 2: <h1>Hello Frodo,</h1><p>Don't forget about the party gifts!</p>
     */
    Destinations: users.map((user) => ({
      Destination: { ToAddresses: [user.emailAddress] },
      ReplacementTemplateData: JSON.stringify({ name: user.firstName }),
    })),
  })),
}
```

```
    DefaultTemplateData: JSON.stringify({ name: "Shireling" }),
    Source: VERIFIED_EMAIL_1,
    Template: templateName,
  });
};

const run = async () => {
  const sendBulkTemplateEmailCommand = createBulkReminderEmailCommand(
    USERS,
    TEMPLATE_NAME,
  );
  try {
    return await sesClient.send(sendBulkTemplateEmailCommand);
  } catch (caught) {
    if (caught instanceof Error && caught.name === "MessageRejected") {
      /** @type { import('@aws-sdk/client-ses').MessageRejected } */
      const messageRejectedError = caught;
      return messageRejectedError;
    }
    throw caught;
  }
};
```

To run the example, enter the following at the command prompt. The email is queued for sending by Amazon SES.

```
node ses_sendbulktemplatedemail.js
```

This example code can be found [here on GitHub](#).

Amazon Simple Notification Service Examples

Amazon Simple Notification Service (Amazon SNS) is a web service that coordinates and manages the delivery or sending of messages to subscribing endpoints or clients.

In Amazon SNS, there are two types of clients—publishers and subscribers—also referred to as producers and consumers.



Publishers communicate asynchronously with subscribers by producing and sending a message to a topic, which is a logical access point and communication channel. Subscribers (web servers, email addresses, Amazon SQS queues, AWS Lambda functions) consume or receive the message or notification over one of the supported protocols (Amazon SQS, HTTP/S, email, SMS, AWS Lambda) when they are subscribed to the topic.

The JavaScript API for Amazon SNS is exposed through the [Class: SNS](#).

Topics

- [Managing Topics in Amazon SNS](#)
- [Publishing Messages in Amazon SNS](#)
- [Managing Subscriptions in Amazon SNS](#)
- [Sending SMS Messages with Amazon SNS](#)

Managing Topics in Amazon SNS



This Node.js code example shows:

- How to create topics in Amazon SNS to which you can publish notifications.
- How to delete topics created in Amazon SNS.
- How to get a list of available topics.
- How to get and set topic attributes.

The Scenario

In this example, you use a series of Node.js modules to create, list, and delete Amazon SNS topics, and to handle topic attributes. The Node.js modules use the SDK for JavaScript to manage topics using these methods of the SNS client class:

- [CreateTopicCommand](#)
- [ListTopicsCommand](#)
- [DeleteTopicCommand](#)
- [GetTopicAttributesCommand](#)
- [SetTopicAttributesCommand](#)

Prerequisite Tasks

To set up and run this example, you must first complete these tasks:

- Set up the project environment to run these Node TypeScript examples, and install the required AWS SDK for JavaScript and third-party modules. Follow the instructions on [GitHub](#).
- Create a shared configurations file with your user credentials. For more information about providing a shared credentials file, see [Shared config and credentials files](#) in the *AWS SDKs and Tools Reference Guide*.

Important

These examples demonstrate how to import/export client service objects and command using ECMAScript6 (ES6).

- This requires Node.js version 13.x or higher. To download and install the latest version of Node.js, see [Node.js downloads](#).
- If you prefer to use CommonJS syntax, see [JavaScript ES6/CommonJS syntax](#).

Creating a Topic

In this example, use a Node.js module to create an Amazon SNS topic.

Create a `libs` directory, and create a Node.js module with the file name `snsClient.js`. Copy and paste the code below into it, which creates the Amazon SNS client object. Replace *REGION* with your AWS Region.

```
import { SNSClient } from "@aws-sdk/client-sns";

// The AWS Region can be provided here using the `region` property. If you leave it
// blank
// the SDK will default to the region set in your AWS config.
export const snsClient = new SNSClient({});
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `create-topic.js`. Configure the SDK as previously shown, including installing the required clients and packages.

Create an object to pass the Name for the new topic to the `CreateTopicCommand` method of the SNS client class. To call the `CreateTopicCommand` method, create an asynchronous function invoking an Amazon SNS service object, passing the parameters object. The data returned contains the ARN of the topic.

Note

Replace *TOPIC_NAME* with the name of the topic.

```
import { CreateTopicCommand } from "@aws-sdk/client-sns";
import { snsClient } from "../libs/snsClient.js";

/**
 * @param {string} topicName - The name of the topic to create.
 */
export const createTopic = async (topicName = "TOPIC_NAME") => {
  const response = await snsClient.send(
    new CreateTopicCommand({ Name: topicName }),
  );
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: '087b8ad2-4593-50c4-a496-d7e90b82cf3e',
```

```
//     extendedRequestId: undefined,  
//     cfId: undefined,  
//     attempts: 1,  
//     totalRetryDelay: 0  
//   },  
//   TopicArn: 'arn:aws:sns:us-east-1:xxxxxxxxxxxx:TOPIC_NAME'  
// }  
return response;  
};
```

To run the example, enter the following at the command prompt.

```
node create-topic.js
```

This example code can be found [here on GitHub](#).

Listing Your Topics

In this example, use a Node.js module to list all Amazon SNS topics.

Create a `libs` directory, and create a Node.js module with the file name `snsClient.js`. Copy and paste the code below into it, which creates the Amazon SNS client object. Replace **REGION** with your AWS Region.

```
import { SNSClient } from "@aws-sdk/client-sns";  
  
// The AWS Region can be provided here using the `region` property. If you leave it  
// blank  
// the SDK will default to the region set in your AWS config.  
export const snsClient = new SNSClient({});
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `list-topics.js`. Configure the SDK as previously shown, including installing the required clients and packages.

Create an empty object to pass to the `ListTopicsCommand` method of the SNS client class. To call the `ListTopicsCommand` method, create an asynchronous function invoking an Amazon SNS service object, passing the parameters object. The data returned contains an array of your topic Amazon Resource Names (ARNs).


```
import { ListTopicsCommand } from "@aws-sdk/client-sns";
import { snsClient } from "../libs/snsClient.js";

export const listTopics = async () => {
  const response = await snsClient.send(new ListTopicsCommand({}));
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: '936bc5ad-83ca-53c2-b0b7-9891167b909e',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  //   Topics: [ { TopicArn: 'arn:aws:sns:us-east-1:xxxxxxxxxxxx:mytopic' } ]
  // }
  return response;
};
```

To run the example, enter the following at the command prompt.

```
node list-topics.js
```

This sample code can be found [here on GitHub](#).

Deleting a Topic

In this example, use a Node.js module to delete an Amazon SNS topic.

Create a `libs` directory, and create a Node.js module with the file name `snsClient.js`. Copy and paste the code below into it, which creates the Amazon SNS client object. Replace **REGION** with your AWS Region.

```
import { SNSClient } from "@aws-sdk/client-sns";

// The AWS Region can be provided here using the `region` property. If you leave it
// blank
// the SDK will default to the region set in your AWS config.
export const snsClient = new SNSClient({});
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `delete-topic.js`. Configure the SDK as previously shown, including installing the required clients and packages.

Create an object containing the `TopicArn` of the topic to delete to pass to the `DeleteTopicCommand` method of the SNS client class. To call the `DeleteTopicCommand` method, create an asynchronous function invoking an Amazon SNS client service object, passing the parameters object.

Note

Replace *TOPIC_ARN* with the Amazon Resource Name (ARN) of the topic you are deleting.

```
import { DeleteTopicCommand } from "@aws-sdk/client-sns";
import { snsClient } from "../libs/snsClient.js";

/**
 * @param {string} topicArn - The ARN of the topic to delete.
 */
export const deleteTopic = async (topicArn = "TOPIC_ARN") => {
  const response = await snsClient.send(
    new DeleteTopicCommand({ TopicArn: topicArn }),
  );
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: 'a10e2886-5a8f-5114-af36-75bd39498332',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   }
  // }
};
```

To run the example, enter the following at the command prompt.

```
node delete-topic.js
```

This example code can be found [here on GitHub](#).

Getting Topic Attributes

In this example, use a Node.js module to retrieve attributes of an Amazon SNS topic.

Create a `libs` directory, and create a Node.js module with the file name `snsClient.js`. Copy and paste the code below into it, which creates the Amazon SNS client object. Replace *REGION* with your AWS Region.

```
import { SNSClient } from "@aws-sdk/client-sns";

// The AWS Region can be provided here using the `region` property. If you leave it
// blank
// the SDK will default to the region set in your AWS config.
export const snsClient = new SNSClient({});
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `get-topic-attributes.js`. Configure the SDK as previously shown.

Create an object containing the `TopicArn` of a topic to delete to pass to the `GetTopicAttributesCommand` method of the SNS client class. To call the `GetTopicAttributesCommand` method, invoking an Amazon SNS client service object, passing the parameters object.

Note

Replace *TOPIC_ARN* with the ARN of the topic.

```
import { GetTopicAttributesCommand } from "@aws-sdk/client-sns";
import { snsClient } from "../libs/snsClient.js";

/**
 * @param {string} topicArn - The ARN of the topic to retrieve attributes for.
 */
export const getTopicAttributes = async (topicArn = "TOPIC_ARN") => {
  const response = await snsClient.send(
    new GetTopicAttributesCommand({
      TopicArn: topicArn,
    }),
  ),
```

```
);
console.log(response);
// {
//   '$metadata': {
//     httpStatusCode: 200,
//     requestId: '36b6a24e-5473-5d4e-ac32-ff72d9a73d94',
//     extendedRequestId: undefined,
//     cfId: undefined,
//     attempts: 1,
//     totalRetryDelay: 0
//   },
//   Attributes: {
//     Policy: '{...}',
//     Owner: 'xxxxxxxxxxxxx',
//     SubscriptionsPending: '1',
//     TopicArn: 'arn:aws:sns:us-east-1:xxxxxxxxxxxx:mytopic',
//     TracingConfig: 'PassThrough',
//     EffectiveDeliveryPolicy: '{"http":{"defaultHealthyRetryPolicy":
{"minDelayTarget":20,"maxDelayTarget":20,"numRetries":3,"numMaxDelayRetries":0,"numNoDelayRetri
{"headerContentType":"text/plain; charset=UTF-8"}}}',
//     SubscriptionsConfirmed: '0',
//     DisplayName: '',
//     SubscriptionsDeleted: '1'
//   }
// }
return response;
};
```

To run the example, enter the following at the command prompt.

```
node get-topic-attributes.js
```

This example code can be found [here on GitHub](#).

Setting Topic Attributes

In this example, use a Node.js module to set the mutable attributes of an Amazon SNS topic.

Create a `libs` directory, and create a Node.js module with the file name `snsClient.js`. Copy and paste the code below into it, which creates the Amazon SNS client object. Replace *REGION* with your AWS Region.

```
import { SNSClient } from "@aws-sdk/client-sns";
```

```
// The AWS Region can be provided here using the `region` property. If you leave it
  blank
// the SDK will default to the region set in your AWS config.
export const snsClient = new SNSClient({});
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `set-topic-attributes.js`. Configure the SDK as previously shown.

Create an object containing the parameters for the attribute update, including the `TopicArn` of the topic whose attributes you want to set, the name of the attribute to set, and the new value for that attribute. You can set only the `Policy`, `DisplayName`, and `DeliveryPolicy` attributes. Pass the parameters to the `SetTopicAttributesCommand` method of the SNS client class. To call the `SetTopicAttributesCommand` method, create an asynchronous function invoking an Amazon SNS client service object, passing the parameters object.

Note

Replace *ATTRIBUTE_NAME* with the name of the attribute you are setting, *TOPIC_ARN* with the Amazon Resource Name (ARN) of the topic whose attributes you want to set, and *NEW_ATTRIBUTE_VALUE* with the new value for that attribute.

```
import { SetTopicAttributesCommand } from "@aws-sdk/client-sns";
import { snsClient } from "../libs/snsClient.js";

export const setTopicAttributes = async (
  topicArn = "TOPIC_ARN",
  attributeName = "DisplayName",
  attributeValue = "Test Topic",
) => {
  const response = await snsClient.send(
    new SetTopicAttributesCommand({
      AttributeName: attributeName,
      AttributeValue: attributeValue,
      TopicArn: topicArn,
    }),
  );
  console.log(response);
```

```
// {  
//   '$metadata': {  
//     httpStatusCode: 200,  
//     requestId: 'd1b08d0e-e9a4-54c3-b8b1-d03238d2b935',  
//     extendedRequestId: undefined,  
//     cfId: undefined,  
//     attempts: 1,  
//     totalRetryDelay: 0  
//   }  
// }  
return response;  
};
```

To run the example, enter the following at the command prompt.

```
node set-topic-attributes.js
```

This example code can be found [here on GitHub](#).

Publishing Messages in Amazon SNS



This Node.js code example shows:

- How to publish messages to an Amazon SNS topic.

The Scenario

In this example, you use a series of Node.js modules to publish messages from Amazon SNS to topic endpoints, emails, or phone numbers. The Node.js modules use the SDK for JavaScript to send messages using this method of the SNS client class:

- [PublishCommand](#)

Prerequisite Tasks

To set up and run this example, you must first complete these tasks:

- Set up the project environment to run these Node TypeScript examples, and install the required AWS SDK for JavaScript and third-party modules. Follow the instructions on [GitHub](#).
- Create a shared configurations file with your user credentials. For more information about providing a shared credentials file, see [Shared config and credentials files](#) in the *AWS SDKs and Tools Reference Guide*.

Important

These examples demonstrate how to import/export client service objects and command using ECMAScript6 (ES6).

- This requires Node.js version 13.x or higher. To download and install the latest version of Node.js, see [Node.js downloads](#).
- If you prefer to use CommonJS syntax, see [JavaScript ES6/CommonJS syntax](#).

Publishing a Message to an SNS Topic

In this example, use a Node.js module to publish a message to an Amazon SNS topic.

Create a `libs` directory, and create a Node.js module with the file name `snsClient.js`. Copy and paste the code below into it, which creates the Amazon SNS client object. Replace **REGION** with your AWS Region.

```
import { SNSClient } from "@aws-sdk/client-sns";

// The AWS Region can be provided here using the `region` property. If you leave it
// blank
// the SDK will default to the region set in your AWS config.
export const snsClient = new SNSClient({});
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `publish-topic.js`. Configure the SDK as previously shown.

Create an object containing the parameters for publishing a message, including the message text and the Amazon Resource Name (ARN) of the Amazon SNS topic. For details on available SMS attributes, see [SetSMSAttributes](#).

Pass the parameters to the `PublishCommand` method of the SNS client class. create an asynchronous function invoking an Amazon SNS client service object, passing the parameters object.

Note

Replace *MESSAGE_TEXT* with the message text, and *TOPIC_ARN* with the ARN of the SNS topic.

```
import { PublishCommand } from "@aws-sdk/client-sns";
import { snsClient } from "../libs/snsClient.js";

/**
 * @param {string | Record<string, any>} message - The message to send. Can be a plain
 * string or an object
 *
 *                                     if you are using the `json`
 * `MessageStructure`.
 * @param {string} topicArn - The ARN of the topic to which you would like to publish.
 */
export const publish = async (
  message = "Hello from SNS!",
  topicArn = "TOPIC_ARN",
) => {
  const response = await snsClient.send(
    new PublishCommand({
      Message: message,
      TopicArn: topicArn,
    }),
  );
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: 'e7f77526-e295-5325-9ee4-281a43ad1f05',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  //   MessageId: 'xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxxxxx'
  // }
}
```



```
    return response;
};
```

To run the example, enter the following at the command prompt.

```
node publish-topic.js
```

This example code can be found [here on GitHub](#).

Managing Subscriptions in Amazon SNS



This Node.js code example shows:

- How to list all subscriptions to an Amazon SNS topic.
- How to subscribe an email address, an application endpoint, or an AWS Lambda function to an Amazon SNS topic.
- How to unsubscribe from Amazon SNS topics.

The Scenario

In this example, you use a series of Node.js modules to publish notification messages to Amazon SNS topics. The Node.js modules use the SDK for JavaScript to manage topics using these methods of the SNS client class:

- [ListSubscriptionsByTopicCommand](#)
- [SubscribeCommand](#)
- [ConfirmSubscriptionCommand](#)
- [UnsubscribeCommand](#)

Prerequisite Tasks

To set up and run this example, you must first complete these tasks:

- Set up the project environment to run these Node TypeScript examples, and install the required AWS SDK for JavaScript and third-party modules. Follow the instructions on [GitHub](#).
- Create a shared configurations file with your user credentials. For more information about providing a shared credentials file, see [Shared config and credentials files](#) in the *AWS SDKs and Tools Reference Guide*.

Important

These examples demonstrate how to import/export client service objects and command using ECMAScript6 (ES6).

- This requires Node.js version 13.x or higher. To download and install the latest version of Node.js, see [Node.js downloads](#).
- If you prefer to use CommonJS syntax, see [JavaScript ES6/CommonJS syntax](#).

Listing Subscriptions to a Topic

In this example, use a Node.js module to list all subscriptions to an Amazon SNS topic.

Create a `libs` directory, and create a Node.js module with the file name `snsClient.js`. Copy and paste the code below into it, which creates the Amazon SNS client object. Replace **REGION** with your AWS Region.

```
import { SNSClient } from "@aws-sdk/client-sns";

// The AWS Region can be provided here using the `region` property. If you leave it
// blank
// the SDK will default to the region set in your AWS config.
export const snsClient = new SNSClient({});
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `list-subscriptions-by-topic.js`. Configure the SDK as previously shown.

Create an object containing the `TopicArn` parameter for the topic whose subscriptions you want to list. Pass the parameters to the `ListSubscriptionsByTopicCommand` method of the SNS

client class. To call the `ListSubscriptionsByTopicCommand` method, create an asynchronous function invoking an Amazon SNS client service object, and passing the parameters object.

Note

Replace *TOPIC_ARN* with the Amazon Resource Name (ARN) for the topic whose subscriptions you want to list .

```
import { ListSubscriptionsByTopicCommand } from "@aws-sdk/client-sns";
import { snsClient } from "../libs/snsClient.js";

/**
 * @param {string} topicArn - The ARN of the topic for which you wish to list
 * subscriptions.
 */
export const listSubscriptionsByTopic = async (topicArn = "TOPIC_ARN") => {
  const response = await snsClient.send(
    new ListSubscriptionsByTopicCommand({ TopicArn: topicArn }),
  );

  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: '0934fedf-0c4b-572e-9ed2-a3e38fadb0c8',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  //   Subscriptions: [
  //     {
  //       SubscriptionArn: 'PendingConfirmation',
  //       Owner: '901487484989',
  //       Protocol: 'email',
  //       Endpoint: 'corepyle@amazon.com',
  //       TopicArn: 'arn:aws:sns:us-east-1:901487484989:mytopic'
  //     }
  //   ]
  // }
  return response;
}
```

```
};
```

To run the example, enter the following at the command prompt.

```
node list-subscriptions-by-topic.js
```

This example code can be found [here on GitHub](#).

Subscribing an Email Address to a Topic

In this example, use a Node.js module to subscribe an email address so that it receives SMTP email messages from an Amazon SNS topic.

Create a `libs` directory, and create a Node.js module with the file name `snsClient.js`. Copy and paste the code below into it, which creates the Amazon SNS client object. Replace **REGION** with your AWS Region.

```
import { SNSClient } from "@aws-sdk/client-sns";

// The AWS Region can be provided here using the `region` property. If you leave it
// blank
// the SDK will default to the region set in your AWS config.
export const snsClient = new SNSClient({});
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `subscribe-email.js`. Configure the SDK as previously shown.

Create an object containing the `Protocol` parameter to specify the email protocol, the `TopicArn` for the topic to subscribe to, and an email address as the message `Endpoint`. Pass the parameters to the `SubscribeCommand` method of the SNS client class. You can use the `subscribe` method to subscribe several different endpoints to an Amazon SNS topic, depending on the values used for parameters passed, as other examples in this topic will show.

To call the `SubscribeCommand` method, create an asynchronous function invoking an Amazon SNS client service object, and passing the parameters object.

Note

Replace *TOPIC_ARN* with the Amazon Resource Name (ARN) for the topic, and *EMAIL_ADDRESS* with the email address to subscribe to.

```
import { SubscribeCommand } from "@aws-sdk/client-sns";
import { snsClient } from "../libs/snsClient.js";

/**
 * @param {string} topicArn - The ARN of the topic for which you wish to confirm a
 * subscription.
 * @param {string} emailAddress - The email address that is subscribed to the topic.
 */
export const subscribeEmail = async (
  topicArn = "TOPIC_ARN",
  emailAddress = "usern@me.com",
) => {
  const response = await snsClient.send(
    new SubscribeCommand({
      Protocol: "email",
      TopicArn: topicArn,
      Endpoint: emailAddress,
    }),
  );
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: 'c8e35bcd-b3c0-5940-9f66-06f6fcc108f0',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  //   SubscriptionArn: 'pending confirmation'
  // }
};
```

To run the example, enter the following at the command prompt.

```
node subscribe-email.js
```

This example code can be found [here on GitHub](#).

Confirming Subscriptions

In this example, use a Node.js module to verify an endpoint owner's intent to receive emails by validating the token sent to the endpoint by a previous subscribe action.

Create a `libs` directory, and create a Node.js module with the file name `snsClient.js`. Copy and paste the code below into it, which creates the Amazon SNS client object. Replace **REGION** with your AWS Region.

```
import { SNSClient } from "@aws-sdk/client-sns";

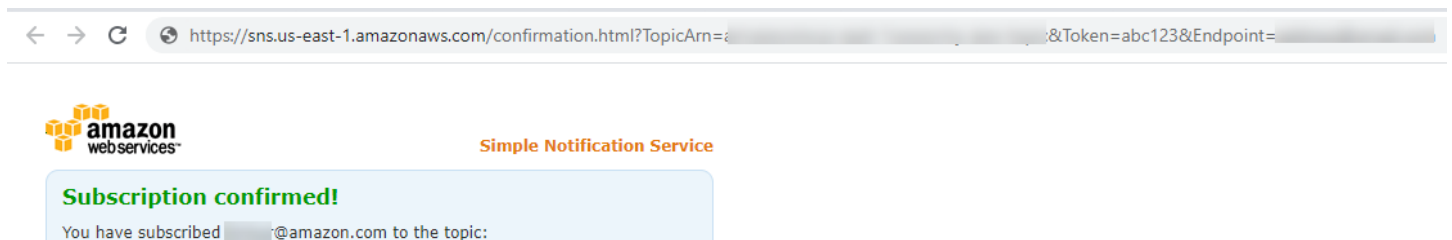
// The AWS Region can be provided here using the `region` property. If you leave it
// blank
// the SDK will default to the region set in your AWS config.
export const snsClient = new SNSClient({});
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `confirm-subscription.js`. Configure the SDK as previously shown, including installing the required clients and packages.

Define the parameters, including the `TOPIC_ARN` and `TOKEN`, and define a value of `TRUE` or `FALSE` for `AuthenticateOnUnsubscribe`.

The token is a short-lived token sent to the owner of an endpoint during a previous `SUBSCRIBE` action. For example, for an email endpoint the `TOKEN` is in the URL of the Confirm Subscription email sent to the email owner. For example, `abc123` is the token in the following URL.



To call the `ConfirmSubscriptionCommand` method, create an asynchronous function invoking an Amazon SNS client service object, passing the parameters object.

Note

Replace *TOPIC_ARN* with the Amazon Resource Name (ARN) for the topic, *TOKEN* with the token value from the URL sent to the endpoint owner in a previous Subscribe action, and define *AuthenticateOnUnsubscribe*. with a value of TRUE or FALSE.

```
import { ConfirmSubscriptionCommand } from "@aws-sdk/client-sns";
import { snsClient } from "../libs/snsClient.js";

/**
 * @param {string} token - This token is sent the subscriber. Only subscribers
 *                        that are not AWS services (HTTP/S, email) need to be
 *                        confirmed.
 * @param {string} topicArn - The ARN of the topic for which you wish to confirm a
 *                            subscription.
 */
export const confirmSubscription = async (
  token = "TOKEN",
  topicArn = "TOPIC_ARN",
) => {
  const response = await snsClient.send(
    // A subscription only needs to be confirmed if the endpoint type is
    // HTTP/S, email, or in another AWS account.
    new ConfirmSubscriptionCommand({
      Token: token,
      TopicArn: topicArn,
      // If this is true, the subscriber cannot unsubscribe while unauthenticated.
      AuthenticateOnUnsubscribe: "false",
    }),
  );
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: '4bb5bce9-805a-5517-8333-e1d2cface90b',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  // }
```

```
// SubscriptionArn: 'arn:aws:sns:us-east-1:xxxxxxxxxxxx:TOPIC_NAME:xxxxxxxx-xxxx-
xxxx-xxxx-xxxxxxxxxxxxx'
// }
return response;
};
```

To run the example, enter the following at the command prompt.

```
node confirm-subscription.js
```

This example code can be found [here on GitHub](#).

Subscribing an Application Endpoint to a Topic

In this example, use a Node.js module to subscribe a mobile application endpoint so it receives notifications from an Amazon SNS topic.

Create a `libs` directory, and create a Node.js module with the file name `snsClient.js`. Copy and paste the code below into it, which creates the Amazon SNS client object. Replace *REGION* with your AWS Region.

```
import { SNSClient } from "@aws-sdk/client-sns";

// The AWS Region can be provided here using the `region` property. If you leave it
// blank
// the SDK will default to the region set in your AWS config.
export const snsClient = new SNSClient({});
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `subscribe-app.js`. Configure the SDK as previously shown, including installing the required modules and packages.

Create an object containing the `Protocol` parameter to specify the application protocol, the `TopicArn` for the topic to subscribe to, and the Amazon Resource Name (ARN) of a mobile application endpoint for the `Endpoint` parameter. Pass the parameters to the `SubscribeCommand` method of the SNS client class.

To call the `SubscribeCommand` method, create an asynchronous function invoking an Amazon SNS service object, passing the parameters object.

Note

Replace *TOPIC_ARN* with the Amazon Resource Name (ARN) for the topic, and *MOBILE_ENDPOINT_ARN* with the endpoint you are subscribing to the topic.

```
import { SubscribeCommand } from "@aws-sdk/client-sns";
import { snsClient } from "../libs/snsClient.js";

/**
 * @param {string} topicArn - The ARN of the topic the subscriber is subscribing to.
 * @param {string} endpoint - The Endpoint ARN of an application. This endpoint is
 * created
 *
 * when an application registers for notifications.
 */
export const subscribeApp = async (
  topicArn = "TOPIC_ARN",
  endpoint = "ENDPOINT",
) => {
  const response = await snsClient.send(
    new SubscribeCommand({
      Protocol: "application",
      TopicArn: topicArn,
      Endpoint: endpoint,
    }),
  );
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: 'c8e35bcd-b3c0-5940-9f66-06f6fcc108f0',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  //   SubscriptionArn: 'pending confirmation'
  // }
  return response;
};
```

To run the example, enter the following at the command prompt.

```
node subscribe-app.js
```

This example code can be found [here on GitHub](#).

Subscribing a Lambda Function to a Topic

In this example, use a Node.js module to subscribe an AWS Lambda function so it receives notifications from an Amazon SNS topic.

Create a `libs` directory, and create a Node.js module with the file name `snsClient.js`. Copy and paste the code below into it, which creates the Amazon SNS client object. Replace **REGION** with your AWS Region.

```
import { SNSClient } from "@aws-sdk/client-sns";

// The AWS Region can be provided here using the `region` property. If you leave it
// blank
// the SDK will default to the region set in your AWS config.
export const snsClient = new SNSClient({});
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `subscribe-lambda.js`. Configure the SDK as previously shown.

Create an object containing the `Protocol` parameter, specifying the `lambda` protocol, the `TopicArn` for the topic to subscribe to, and the Amazon Resource Name (ARN) of an AWS Lambda function as the `Endpoint` parameter. Pass the parameters to the `SubscribeCommand` method of the SNS client class.

To call the `SubscribeCommand` method, create an asynchronous function invoking an Amazon SNS client service object, passing the parameters object.

Note

Replace **TOPIC_ARN** with the Amazon Resource Name (ARN) for the topic, and **LAMBDA_FUNCTION_ARN** with the Amazon Resource Name (ARN) of the Lambda function.

```
import { SubscribeCommand } from "@aws-sdk/client-sns";
```

```
import { snsClient } from "../libs/snsClient.js";

/**
 * @param {string} topicArn - The ARN of the topic the subscriber is subscribing to.
 * @param {string} endpoint - The Endpoint ARN of and AWS Lambda function.
 */
export const subscribeLambda = async (
  topicArn = "TOPIC_ARN",
  endpoint = "ENDPOINT",
) => {
  const response = await snsClient.send(
    new SubscribeCommand({
      Protocol: "lambda",
      TopicArn: topicArn,
      Endpoint: endpoint,
    }),
  );
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: 'c8e35bcd-b3c0-5940-9f66-06f6fcc108f0',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  //   SubscriptionArn: 'pending confirmation'
  // }
  return response;
};
```

To run the example, enter the following at the command prompt.

```
node subscribe-lambda.js
```

This example code can be found [here on GitHub](#).

Unsubscribing from a Topic

In this example, use a Node.js module to unsubscribe an Amazon SNS topic subscription.

Create a `libs` directory, and create a Node.js module with the file name `snsClient.js`. Copy and paste the code below into it, which creates the Amazon SNS client object. Replace *REGION* with your AWS Region.

```
import { SNSClient } from "@aws-sdk/client-sns";

// The AWS Region can be provided here using the `region` property. If you leave it
// blank
// the SDK will default to the region set in your AWS config.
export const snsClient = new SNSClient({});
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `unsubscribe.js`. Configure the SDK as previously shown, including installing the required clients and packages.

Create an object containing the `SubscriptionArn` parameter, specifying the Amazon Resource Name (ARN) of the subscription to unsubscribe. Pass the parameters to the `UnsubscribeCommand` method of the SNS client class.

To call the `UnsubscribeCommand` method, create an asynchronous function invoking an Amazon SNS client service object, passing the parameters object.

Note

Replace *TOPIC_SUBSCRIPTION_ARN* with the Amazon Resource Name (ARN) of the subscription to unsubscribe.

```
import { UnsubscribeCommand } from "@aws-sdk/client-sns";
import { snsClient } from "../libs/snsClient.js";

/**
 * @param {string} subscriptionArn - The ARN of the subscription to cancel.
 */
const unsubscribe = async (
  subscriptionArn = "arn:aws:sns:us-east-1:xxxxxxxxxxxx:mytopic:xxxxxxxx-xxxx-xxxx-
  xxxx-xxxxxxxxxxxx",
) => {
  const response = await snsClient.send(
    new UnsubscribeCommand({
```

```
        SubscriptionArn: subscriptionArn,
    }),
  );
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: '0178259a-9204-507c-b620-78a7570a44c6',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   }
  // }
  return response;
};
```

To run the example, enter the following at the command prompt.

```
node unsubscribe.js
```

This example code can be found [here on GitHub](#).

Sending SMS Messages with Amazon SNS



This Node.js code example shows:

- How to get and set SMS messaging preferences for Amazon SNS.
- How to check a phone number to see if it has opted out of receiving SMS messages.
- How to get a list of phone numbers that have opted out of receiving SMS messages.
- How to send an SMS message.

The Scenario

You can use Amazon SNS to send text messages, or SMS messages, to SMS-enabled devices. You can send a message directly to a phone number, or you can send a message to multiple phone

numbers at once by subscribing those phone numbers to a topic and sending your message to the topic.

In this example, you use a series of Node.js modules to publish SMS text messages from Amazon SNS to SMS-enabled devices. The Node.js modules use the SDK for JavaScript to publish SMS messages using these methods of the SNS client class:

- [GetSMSAttributesCommand](#)
- [SetSMSAttributesCommand](#)
- [CheckIfPhoneNumberIsOptedOutCommand](#)
- [ListPhoneNumbersOptedOutCommand](#)
- [PublishCommand](#)

Prerequisite Tasks

To set up and run this example, you must first complete these tasks:

- Set up the project environment to run these Node TypeScript examples, and install the required AWS SDK for JavaScript and third-party modules. Follow the instructions on [GitHub](#).
- Create a shared configurations file with your user credentials. For more information about providing a shared credentials file, see [Shared config and credentials files](#) in the *AWS SDKs and Tools Reference Guide*.

Important

These examples demonstrate how to import/export client service objects and command using ECMAScript6 (ES6).

- This requires Node.js version 13.x or higher. To download and install the latest version of Node.js, see [Node.js downloads](#).
- If you prefer to use CommonJS syntax, see [JavaScript ES6/CommonJS syntax](#).

Getting SMS Attributes

Use Amazon SNS to specify preferences for SMS messaging, such as how your deliveries are optimized (for cost or for reliable delivery), your monthly spending limit, how message deliveries

are logged, and whether to subscribe to daily SMS usage reports. These preferences are retrieved and set as SMS attributes for Amazon SNS.

In this example, use a Node.js module to get the current SMS attributes in Amazon SNS.

Create a `libs` directory, and create a Node.js module with the file name `snsClient.js`. Copy and paste the code below into it, which creates the Amazon SNS client object. Replace *REGION* with your AWS Region.

```
import { SNSClient } from "@aws-sdk/client-sns";

// The AWS Region can be provided here using the `region` property. If you leave it
// blank
// the SDK will default to the region set in your AWS config.
export const snsClient = new SNSClient({});
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `get-sms-attributes.js`.

Configure the SDK as previously shown, including downloading the required clients and packages. Create an object containing the parameters for getting SMS attributes, including the names of the individual attributes to get. For details on available SMS attributes, see [SetSMSAttributes](#) in the Amazon Simple Notification Service API Reference.

This example gets the `DefaultSMSType` attribute, which controls whether SMS messages are sent as `Promotional`, which optimizes message delivery to incur the lowest cost, or as `Transactional`, which optimizes message delivery to achieve the highest reliability. Pass the parameters to the `SetTopicAttributesCommand` method of the SNS client class. To call the `SetSMSAttributesCommand` method, create an asynchronous function invoking an Amazon SNS client service object, passing the parameters object.

Note

Replace *ATTRIBUTE_NAME* with the name of the attribute.

```
import { GetSMSAttributesCommand } from "@aws-sdk/client-sns";
import { snsClient } from "../libs/snsClient.js";
```

```
export const getSmsAttributes = async () => {
  const response = await snsClient.send(
    // If you have not modified the account-level mobile settings of SNS,
    // the DefaultSMSType is undefined. For this example, it was set to
    // Transactional.
    new GetSMSAttributesCommand({ attributes: ["DefaultSMSType"] }),
  );

  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: '67ad8386-4169-58f1-bdb9-debd281d48d5',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  //   attributes: { DefaultSMSType: 'Transactional' }
  // }
  return response;
};
```

To run the example, enter the following at the command prompt.

```
node get-sms-attributes.js
```

This example code can be found [here on GitHub](#).

Setting SMS Attributes

In this example, use a Node.js module to get the current SMS attributes in Amazon SNS.

Create a `libs` directory, and create a Node.js module with the file name `snsClient.js`. Copy and paste the code below into it, which creates the Amazon SNS client object. Replace **REGION** with your AWS Region.

```
import { SNSClient } from "@aws-sdk/client-sns";

// The AWS Region can be provided here using the `region` property. If you leave it
// blank
// the SDK will default to the region set in your AWS config.
```



```
export const snsClient = new SNSClient({});
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `set-sms-attribute-type.js`. Configure the SDK as previously shown, including installing the required clients and packages. Create an object containing the parameters for setting SMS attributes, including the names of the individual attributes to set and the values to set for each. For details on available SMS attributes, see [SetSMSAttributes](#) in the Amazon Simple Notification Service API Reference.

This example sets the `DefaultSMSType` attribute to `Transactional`, which optimizes message delivery to achieve the highest reliability. Pass the parameters to the `SetTopicAttributesCommand` method of the SNS client class. To call the `SetSMSAttributesCommand` method, create an asynchronous function invoking an Amazon SNS client service object, passing the parameters object.

```
import { SetSMSAttributesCommand } from "@aws-sdk/client-sns";
import { snsClient } from "../libs/snsClient.js";

/**
 * @param {"Transactional" | "Promotional"} defaultSmsType
 */
export const setSmsType = async (defaultSmsType = "Transactional") => {
  const response = await snsClient.send(
    new SetSMSAttributesCommand({
      attributes: {
        // Promotional - (Default) Noncritical messages, such as marketing messages.
        // Transactional - Critical messages that support customer transactions,
        // such as one-time passcodes for multi-factor authentication.
        DefaultSMSType: defaultSmsType,
      },
    }),
  );
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: '1885b977-2d7e-535e-8214-e44be727e265',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   }
  // }
```

```
// }  
// }  
return response;  
};
```

To run the example, enter the following at the command prompt.

```
node set-sms-attribute-type.js
```

This example code can be found [here on GitHub](#).

Checking If a Phone Number Has Opted Out

In this example, use a Node.js module to check a phone number to see if it has opted out from receiving SMS messages.

Create a `libs` directory, and create a Node.js module with the file name `snsClient.js`. Copy and paste the code below into it, which creates the Amazon SNS client object. Replace **REGION** with your AWS Region.

```
import { SNSClient } from "@aws-sdk/client-sns";  
  
// The AWS Region can be provided here using the `region` property. If you leave it  
// blank  
// the SDK will default to the region set in your AWS config.  
export const snsClient = new SNSClient({});
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `check-if-phone-number-is-opted-out.js`. Configure the SDK as previously shown. Create an object containing the phone number to check as a parameter.

This example sets the `PhoneNumber` parameter to specify the phone number to check. Pass the object to the `CheckIfPhoneNumberIsOptedOutCommand` method of the SNS client class. To call the `CheckIfPhoneNumberIsOptedOutCommand` method, create an asynchronous function invoking an Amazon SNS client service object, passing the parameters object.

Note

- 1.

Replace *PHONE_NUMBER* with the phone number.

```
import { CheckIfPhoneNumberIsOptedOutCommand } from "@aws-sdk/client-sns";

import { snsClient } from "../libs/snsClient.js";

export const checkIfPhoneNumberIsOptedOut = async (
  phoneNumber = "5555555555",
) => {
  const command = new CheckIfPhoneNumberIsOptedOutCommand({
    phoneNumber,
  });

  const response = await snsClient.send(command);
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: '3341c28a-cdc8-5b39-a3ee-9fb0ee125732',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  //   isOptedOut: false
  // }
  return response;
};
```

To run the example, enter the following at the command prompt.

```
node check-if-phone-number-is-opted-out.js
```

This example code can be found [here on GitHub](#).

Listing Opted-Out Phone Numbers

In this example, use a Node.js module to get a list of phone numbers that have opted out from receiving SMS messages.

Create a `libs` directory, and create a Node.js module with the file name `snsClient.js`. Copy and paste the code below into it, which creates the Amazon SNS client object. Replace *REGION* with your AWS Region.

```
import { SNSClient } from "@aws-sdk/client-sns";

// The AWS Region can be provided here using the `region` property. If you leave it
// blank
// the SDK will default to the region set in your AWS config.
export const snsClient = new SNSClient({});
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `list-phone-numbers-opted-out.js`. Configure the SDK as previously shown. Create an empty object as a parameter.

Pass the object to the `ListPhoneNumbersOptedOutCommand` method of the SNS client class. To call the `ListPhoneNumbersOptedOutCommand` method, create an asynchronous function invoking an Amazon SNS client service object, passing the parameters object.

```
import { ListPhoneNumbersOptedOutCommand } from "@aws-sdk/client-sns";
import { snsClient } from "../libs/snsClient.js";

export const listPhoneNumbersOptedOut = async () => {
  const response = await snsClient.send(
    new ListPhoneNumbersOptedOutCommand({}),
  );
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: '44ff72fd-1037-5042-ad96-2fc16601df42',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  //   phoneNumbers: ['+15555550100']
  // }
  return response;
};
```

To run the example, enter the following at the command prompt.

```
node list-phone-numbers-opted-out.js
```

This example code can be found [here on GitHub](#).

Publishing an SMS Message

In this example, use a Node.js module to send an SMS message to a phone number.

Create a `libs` directory, and create a Node.js module with the file name `snsClient.js`. Copy and paste the code below into it, which creates the Amazon SNS client object. Replace **REGION** with your AWS Region.

```
import { SNSClient } from "@aws-sdk/client-sns";

// The AWS Region can be provided here using the `region` property. If you leave it
// blank
// the SDK will default to the region set in your AWS config.
export const snsClient = new SNSClient({});
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `publish-sms.js`. Configure the SDK as previously shown, including installing the required clients and packages. Create an object containing the `Message` and `PhoneNumber` parameters.

When you send an SMS message, specify the phone number using the E.164 format. E.164 is a standard for the phone number structure used for international telecommunication. Phone numbers that follow this format can have a maximum of 15 digits, and they are prefixed with the plus character (+) and the country code. For example, a US phone number in E.164 format would appear as `+1001XXX5550100`.

This example sets the `PhoneNumber` parameter to specify the phone number to send the message. Pass the object to the `PublishCommand` method of the SNS client class. To call the `PublishCommand` method, create an asynchronous function invoking an Amazon SNS service object, passing the parameters object.

Note

Replace *TEXT_MESSAGE* with the text message, and *PHONE_NUMBER* with the phone number.

```
import { PublishCommand } from "@aws-sdk/client-sns";
import { snsClient } from "../libs/snsClient.js";

/**
 * @param {string | Record<string, any>} message - The message to send. Can be a plain
 * string or an object
 *
 * if you are using the `json`
 * `MessageStructure`.
 * @param {*} phoneNumber - The phone number to send the message to.
 */
export const publish = async (
  message = "Hello from SNS!",
  phoneNumber = "+15555555555",
) => {
  const response = await snsClient.send(
    new PublishCommand({
      Message: message,
      // One of PhoneNumber, TopicArn, or TargetArn must be specified.
      PhoneNumber: phoneNumber,
    }),
  );
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: '7410094f-efc7-5f52-af03-54737569ab77',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  //   MessageId: 'xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxxxx'
  // }
  return response;
};
```

To run the example, enter the following at the command prompt.

```
node publish-sms.js
```

This example code can be found [here on GitHub](#).

Amazon Transcribe examples

Amazon Transcribe makes it easy for developers to add speech to text capabilities to their applications.



The JavaScript API for Amazon Transcribe is exposed through the [TranscribeService](#) client class.

Topics

- [Amazon Transcribe examples](#)
- [Amazon Transcribe medical examples](#)

Amazon Transcribe examples

In this example, a series of Node.js modules are used to create, list, and delete transcription jobs using the following methods of the TranscribeService client class:

- [StartTranscriptionJobCommand](#)
- [ListTranscriptionJobsCommand](#)
- [DeleteTranscriptionJobCommand](#)

For more information about Amazon Transcribe users, see the [Amazon Transcribe developer guide](#).

Prerequisite tasks

To set up and run this example, you must first complete these tasks:

- Set up the project environment to run these Node TypeScript examples, and install the required AWS SDK for JavaScript and third-party modules. Follow the instructions on [GitHub](#).
- Create a shared configurations file with your user credentials. For more information about providing a shared credentials file, see [Shared config and credentials files](#) in the *AWS SDKs and Tools Reference Guide*.

Important

These examples demonstrate how to import/export client service objects and command using ECMAScript6 (ES6).

- This requires Node.js version 13.x or higher. To download and install the latest version of Node.js, see [Node.js downloads](#).
- If you prefer to use CommonJS syntax, see [JavaScript ES6/CommonJS syntax](#)

Starting an Amazon Transcribe job

This example demonstrates how to start a Amazon Transcribe transcription job using the AWS SDK for JavaScript. For more information, see [StartTranscriptionJobCommand](#).

Create a `libs` directory, and create a Node.js module with the file name `transcribeClient.js`. Copy and paste the code below into it, which creates the Amazon Transcribe client object. Replace **REGION** with your AWS Region.

```
import { TranscribeClient } from "@aws-sdk/client-transcribe";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon Transcribe service client object.
const transcribeClient = new TranscribeClient({ region: REGION });
export { transcribeClient };
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `transcribe-create-job.js`. Make sure to configure the SDK as previously shown, including installing the required clients and packages. Create a parameters object, specifying the required parameters. Start the job using the `StartMedicalTranscriptionJobCommand` command.

Note

Replace *MEDICAL_JOB_NAME* with a name for the transcription job. For *OUTPUT_BUCKET_NAME* specify the Amazon S3 bucket where the output is saved. For *JOB_TYPE* specify types of job. For *SOURCE_LOCATION* specify the location of the source file. For *SOURCE_FILE_LOCATION* specify the location of the input media file.

```
// Import the required AWS SDK clients and commands for Node.js
import { StartTranscriptionJobCommand } from "@aws-sdk/client-transcribe";
import { transcribeClient } from "../libs/transcribeClient.js";

// Set the parameters
export const params = {
  TranscriptionJobName: "JOB_NAME",
  LanguageCode: "LANGUAGE_CODE", // For example, 'en-US'
  MediaFormat: "SOURCE_FILE_FORMAT", // For example, 'wav'
  Media: {
    MediaFileUri: "SOURCE_LOCATION",
    // For example, "https://transcribe-demo.s3-REGION.amazonaws.com/hello_world.wav"
  },
  OutputBucketName: "OUTPUT_BUCKET_NAME"
};

export const run = async () => {
  try {
    const data = await transcribeClient.send(
      new StartTranscriptionJobCommand(params)
    );
    console.log("Success - put", data);
    return data; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};

run();
```

To run the example, enter the following at the command prompt.

```
node transcribe-create-job.js
```

This sample code can be found [here on GitHub](#).

List Amazon Transcribe jobs

This example shows how list the Amazon Transcribe transcription jobs using the AWS SDK for JavaScript. For more information about what other setting you can modify, see [ListTranscriptionJobCommand](#).

Create a `libs` directory, and create a Node.js module with the file name `transcribeClient.js`. Copy and paste the code below into it, which creates the Amazon Transcribe client object. Replace **REGION** with your AWS Region.

```
import { TranscribeClient } from "@aws-sdk/client-transcribe";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon Transcribe service client object.
const transcribeClient = new TranscribeClient({ region: REGION });
export { transcribeClient };
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `transcribe-list-jobs.js`. Make sure to configure the SDK as previously shown, including installing the required clients and packages. Create a parameters object with the required parameters.

Note

Replace **KEY_WORD** with a keyword that the returned jobs name must contain.

```
// Import the required AWS SDK clients and commands for Node.js
import { ListTranscriptionJobsCommand } from "@aws-sdk/client-transcribe";
```

```
import { transcribeClient } from "../libs/transcribeClient.js";

// Set the parameters
export const params = {
  JobNameContains: "KEYWORD", // Not required. Returns only transcription
  // job names containing this string
};

export const run = async () => {
  try {
    const data = await transcribeClient.send(
      new ListTranscriptionJobsCommand(params)
    );
    console.log("Success", data.TranscriptionJobSummaries);
    return data; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};

run();
```

To run the example, enter the following at the command prompt.

```
node transcribe-list-jobs.js
```

This sample code can be found [here on GitHub](#).

Deleting a Amazon Transcribe job

This example shows how to delete an Amazon Transcribe transcription job using the AWS SDK for JavaScript. For more information about optional, see [DeleteTranscriptionJobCommand](#).

Create a `libs` directory, and create a Node.js module with the file name `transcribeClient.js`. Copy and paste the code below into it, which creates the Amazon Transcribe client object. Replace **REGION** with your AWS Region.

```
import { TranscribeClient } from "@aws-sdk/client-transcribe";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create Transcribe service object.
const transcribeClient = new TranscribeClient({ region: REGION });
export { transcribeClient };
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `transcribe-delete-job.js`. Make sure to configure the SDK as previously shown, including installing the required clients and packages. Specify the AWS Region, and the name of the job you want to delete.

Note

Replace *JOB_NAME* with the name of the job to delete.

```
// Import the required AWS SDK clients and commands for Node.js
import { DeleteTranscriptionJobCommand } from "@aws-sdk/client-transcribe";
import { transcribeClient } from "../libs/transcribeClient.js";

// Set the parameters
export const params = {
  TranscriptionJobName: "JOB_NAME", // Required. For example, 'transcription_demo'
};

export const run = async () => {
  try {
    const data = await transcribeClient.send(
      new DeleteTranscriptionJobCommand(params)
    );
    console.log("Success - deleted");
    return data; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

To run the example, enter the following at the command prompt.

```
node transcribe-delete-job.js
```

This sample code can be found [here on GitHub](#).

Amazon Transcribe medical examples

In this example, a series of Node.js modules are used to create, list, and delete medical transcription jobs using the following methods of the `TranscribeService` client class:

- [StartMedicalTranscriptionJobCommand](#)
- [ListMedicalTranscriptionJobsCommand](#)
- [DeleteMedicalTranscriptionJobCommand](#)

For more information about Amazon Transcribe users, see the [Amazon Transcribe developer guide](#).

Prerequisite tasks

To set up and run this example, you must first complete these tasks:

- Set up the project environment to run these Node TypeScript examples, and install the required AWS SDK for JavaScript and third-party modules. Follow the instructions on [GitHub](#).
- Create a shared configurations file with your user credentials. For more information about providing a shared credentials file, see [Shared config and credentials files](#) in the *AWS SDKs and Tools Reference Guide*.

Important

These examples demonstrate how to import/export client service objects and command using ECMAScript6 (ES6).

- This requires Node.js version 13.x or higher. To download and install the latest version of Node.js, see [Node.js downloads](#).
- If you prefer to use CommonJS syntax, see [JavaScript ES6/CommonJS syntax](#)

Starting an Amazon Transcribe medical transcription job

This example demonstrates how to start a Amazon Transcribe medical transcription job using the AWS SDK for JavaScript. For more information, see [startMedicalTranscriptionJob](#).

Create a `libs` directory, and create a Node.js module with the file name `transcribeClient.js`. Copy and paste the code below into it, which creates the Amazon Transcribe client object. Replace **REGION** with your AWS Region.

```
import { TranscribeClient } from "@aws-sdk/client-transcribe";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create Transcribe service object.
const transcribeClient = new TranscribeClient({ region: REGION });
export { transcribeClient };
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `transcribe-create-medical-job.js`. Make sure to configure the SDK as previously shown, including installing the required clients and packages. Create a parameters object, specifying the required parameters. Start the medical job using the `StartMedicalTranscriptionJobCommand` command.

Note

Replace **MEDICAL_JOB_NAME** with a name for the medical transcription job. For **OUTPUT_BUCKET_NAME** specify the Amazon S3 bucket where the output is saved. For **JOB_TYPE** specify types of job. For **SOURCE_LOCATION** specify the location of the source file. For **SOURCE_FILE_LOCATION** specify the location of the input media file.

```
// Import the required AWS SDK clients and commands for Node.js
import { StartMedicalTranscriptionJobCommand } from "@aws-sdk/client-transcribe";
import { transcribeClient } from "../libs/transcribeClient.js";

// Set the parameters
export const params = {
  MedicalTranscriptionJobName: "MEDICAL_JOB_NAME", // Required
  OutputBucketName: "OUTPUT_BUCKET_NAME", // Required
  Specialty: "PRIMARYCARE", // Required. Possible values are 'PRIMARYCARE'
  Type: "JOB_TYPE", // Required. Possible values are 'CONVERSATION' and 'DICTATION'
  LanguageCode: "LANGUAGE_CODE", // For example, 'en-US'
  MediaFormat: "SOURCE_FILE_FORMAT", // For example, 'wav'
  Media: {
    MediaFileUri: "SOURCE_FILE_LOCATION",
```

```
// The S3 object location of the input media file. The URI must be in the same
region
// as the API endpoint that you are calling. For example,
// "https://transcribe-demo.s3-REGION.amazonaws.com/hello_world.wav"
},
};

export const run = async () => {
  try {
    const data = await transcribeClient.send(
      new StartMedicalTranscriptionJobCommand(params)
    );
    console.log("Success - put", data);
    return data; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

To run the example, enter the following at the command prompt.

```
node transcribe-create-medical-job.js
```

This sample code can be found [here on GitHub](#).

Listing Amazon Transcribe medical jobs


This example shows how to list the Amazon Transcribe transcription jobs using the AWS SDK for JavaScript. For more information, see [ListTranscriptionMedicalJobsCommand](#).

Create a `libs` directory, and create a Node.js module with the file name `transcribeClient.js`. Copy and paste the code below into it, which creates the Amazon Transcribe client object. Replace **REGION** with your AWS Region.

```
import { TranscribeClient } from "@aws-sdk/client-transcribe";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon Transcribe service client object.
const transcribeClient = new TranscribeClient({ region: REGION });
export { transcribeClient };
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `transcribe-list-medical-jobs.js`. Make sure to configure the SDK as previously shown, including installing the required clients and packages. Create a parameters object with the required parameters, and list the medical jobs using the `ListMedicalTranscriptionJobsCommand` command.

 **Note**

Replace **KEYWORD** with a keyword that the returned jobs name must contain.

```
// Import the required AWS SDK clients and commands for Node.js

import { ListMedicalTranscriptionJobsCommand } from "@aws-sdk/client-transcribe";
import { transcribeClient } from "../libs/transcribeClient.js";

// Set the parameters
export const params = {
  JobNameContains: "KEYWORD", // Returns only transcription job names containing this
  string
};

export const run = async () => {
  try {
    const data = await transcribeClient.send(
      new ListMedicalTranscriptionJobsCommand(params)
    );
    console.log("Success", data.MedicalTranscriptionJobName);
    return data; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

To run the example, enter the following at the command prompt.

```
node transcribe-list-medical-jobs.js
```

This sample code can be found [here on GitHub](#).

Deleting an Amazon Transcribe medical job

This example shows how to delete an Amazon Transcribe transcription job using the AWS SDK for JavaScript. For more information about optional, see [DeleteTranscriptionMedicalJobCommand](#).

Create a `libs` directory, and create a Node.js module with the file name `transcribeClient.js`. Copy and paste the code below into it, which creates the Amazon Transcribe client object. Replace **REGION** with your AWS Region.

```
import { TranscribeClient } from "@aws-sdk/client-transcribe";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create Transcribe service object.
const transcribeClient = new TranscribeClient({ region: REGION });
export { transcribeClient };
```

This example code can be found [here on GitHub](#).

Create a Node.js module with the file name `transcribe-delete-job.js`. Make sure to configure the SDK as previously shown, including installing the required clients and packages. Create a parameters object with the required parameters, and delete the medical job using the `DeleteMedicalJobCommand` command.

Note

Replace **JOB_NAME** with the name of the job to delete.

```
// Import the required AWS SDK clients and commands for Node.js
import { DeleteMedicalTranscriptionJobCommand } from "@aws-sdk/client-transcribe";
import { transcribeClient } from "../libs/transcribeClient.js";

// Set the parameters
export const params = {
  MedicalTranscriptionJobName: "MEDICAL_JOB_NAME", // For example,
  'medical_transcription_demo'
};

export const run = async () => {
  try {
```

```
const data = await transcribeClient.send(
  new DeleteMedicalTranscriptionJobCommand(params)
);
console.log("Success - deleted");
return data; // For unit tests.
} catch (err) {
  console.log("Error", err);
}
};
run();
```

To run the example, enter the following at the command prompt.

```
node transcribe-delete-medical-job.js
```

This sample code can be found [here on GitHub](#).

Setting up Node.js on an Amazon EC2 instance

A common scenario for using Node.js with the SDK for JavaScript is to set up and run a Node.js web application on an Amazon Elastic Compute Cloud (Amazon EC2) instance. In this tutorial, you will create a Linux instance, connect to it using SSH, and then install Node.js to run on that instance.

Prerequisites

This tutorial assumes that you have already launched a Linux instance with a public DNS name that is reachable from the internet and to which you are able to connect using SSH. For more information, see [Step 1: Launch an instance](#) in the *Amazon EC2 User Guide*.

Important

Use the **Amazon Linux 2023** Amazon Machine Image (AMI) when launching a new Amazon EC2 instance.

You must also have configured your security group to allow SSH (port 22), HTTP (port 80), and HTTPS (port 443) connections. For more information about these prerequisites, see [Setting up with Amazon EC2](#) in the *Amazon EC2 User Guide*.

Procedure

The following procedure helps you install Node.js on an Amazon Linux instance. You can use this server to host a Node.js web application.

To set up Node.js on your Linux instance

1. Connect to your Linux instance as `ec2-user` using SSH.
2. Install node version manager (nvm) by typing the following at the command line.

Warning

AWS does not control the following code. Before you run it, be sure to verify its authenticity and integrity. More information about this code can be found in the [nvm](#) GitHub repository.

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.7/install.sh | bash
```

We will use nvm to install Node.js because nvm can install multiple versions of Node.js and allow you to switch between them.

3. Load nvm by typing the following at the command line.

```
source ~/.bashrc
```

4. Use nvm to install the latest LTS version of Node.js by typing the following at the command line.

```
nvm install --lts
```

Installing Node.js also installs the Node Package Manager (npm) so you can install additional modules as needed.

5. Test that Node.js is installed and running correctly by typing the following at the command line.

```
node -e "console.log('Running Node.js ' + process.version)"
```

This displays the following message that shows the version of Node.js that is running.

```
Running Node.js VERSION
```

Note

The node installation only applies to the current Amazon EC2 session. If you restart your CLI session you need to use `nvm` again to enable the installed node version. If the instance is terminated, you need to install node again. The alternative is to make an Amazon Machine Image (AMI) of the Amazon EC2 instance once you have the configuration that you want to keep, as described in the following topic.

Creating an Amazon Machine Image (AMI)

After you install Node.js on an Amazon EC2 instance, you can create an Amazon Machine Image (AMI) from that instance. Creating an AMI makes it easy to provision multiple Amazon EC2 instances with the same Node.js installation. For more information about creating an AMI from an existing instance, see [Creating an amazon EBS-backed Linux AMI](#) in the *Amazon EC2 User Guide*.

Related resources

For more information about the commands and software used in this topic, see the following webpages:

- Node version manager (nvm) –See [nvm repo on GitHub](#).
- Node Package Manager (npm) –See [npm website](#).

Build an app to submit data to DynamoDB

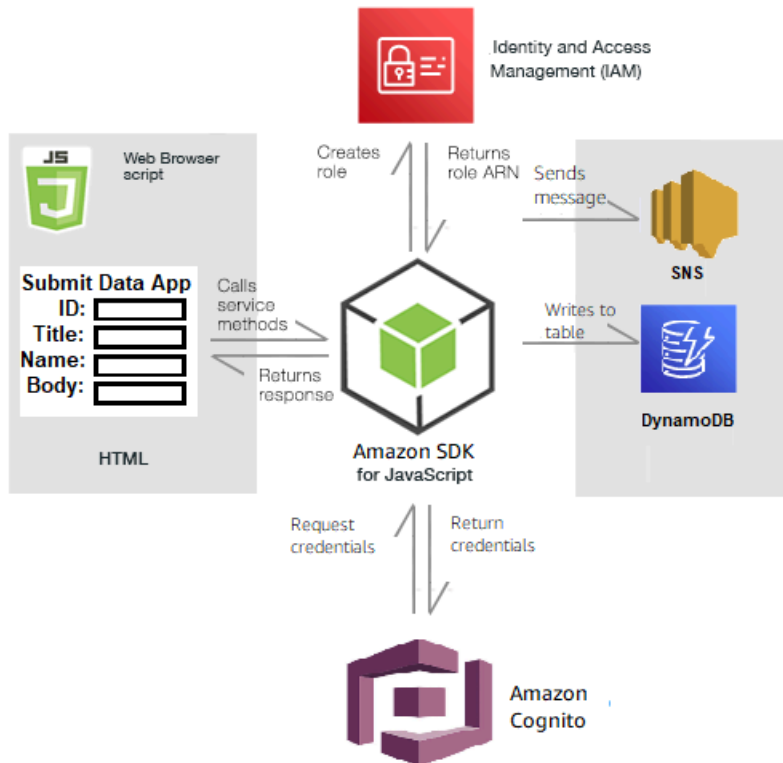
This cross-service Node.js tutorial shows how to build an app that enables users to submit data to an Amazon DynamoDB table. This app uses the following services:

- AWS Identity and Access Management (IAM) and Amazon Cognito for authorization and permissions.
- Amazon DynamoDB (DynamoDB) to create and update the tables.

- Amazon Simple Notification Service (Amazon SNS) to notify the app administrator when a user updates the table.

The scenario

In this tutorial, an HTML page provides a browser-based application for submitting data to a Amazon DynamoDB table. The app uses Amazon SNS to notify the app administrator when a user updates the table.



To build the app:

1. [Prerequisites](#)
2. [Provision resources](#)
3. [Create the HTML](#)
4. [Create the browser script](#)
5. [Next steps](#)

Prerequisites

Complete the following prerequisite tasks:

- Set up the project environment to run these Node TypeScript examples, and install the required AWS SDK for JavaScript and third-party modules. Follow the instructions on [GitHub](#).
- Create a shared configurations file with your user credentials. For more information about providing a shared credentials file, see [Shared config and credentials files](#) in the *AWS SDKs and Tools Reference Guide*.

Create the AWS resources

This app requires the following resources:

- AWS Identity and Access Management (IAM) Unauthenticated Amazon Cognito user role with the following permissions:
 - sns:Publish
 - dynamodb:PutItem
- A DynamoDB table.

You can create these resources manually in the AWS console, but we recommend provisioning these resources using AWS CloudFormation as described in this tutorial.

Create the AWS resources using AWS CloudFormation

AWS CloudFormation enables you to create and provision AWS infrastructure deployments predictably and repeatedly. For more information about AWS CloudFormation, see the [AWS CloudFormation User Guide](#).

To create the AWS CloudFormation stack using the AWS CLI:

1. Install and configure the AWS CLI following the instructions in the [AWS CLI User Guide](#).
2. Create a file named `setup.yaml` in the root directory of your project folder, and copy the content [here on GitHub](#) into it.

Note

The AWS CloudFormation template was generated using the AWS CDK available [here on GitHub](#). For more information about the AWS CDK, see the [AWS Cloud Development Kit \(AWS CDK\) Developer Guide](#).

3. Run the following command from the command line, replacing *STACK_NAME* with a unique name for the stack, and *REGION* in your AWS region.

Important

The stack name must be unique within an AWS Region and AWS account. You can specify up to 128 characters, and numbers and hyphens are allowed.

```
aws cloudformation create-stack --stack-name STACK_NAME --template-body file://
setup.yaml --capabilities CAPABILITY_IAM --region REGION
```

For more information on the `create-stack` command parameters, see the [AWS CLI Command Reference guide](#), and the [AWS CloudFormation User Guide](#).

To view the resources created, open AWS CloudFormation in the AWS management console, choose the stack, and select the **Resources** tab.

4. When the stack is create, use the AWS SDK for JavaScript to populate the DynamoDB table, as described in [Populating the table](#).

Populating the table

To populate the table, first create a directory named `libs`, and in it create a file named `dynamoClient.js`, and paste the content below into it. Replace *REGION* with your AWS Region, and replace *IDENTITY_POOL_ID* with an Amazon Cognito Identity Pool ID. This creates the DynamoDB client object.

```
import { CognitoIdentityClient } from "@aws-sdk/client-cognito-identity";
import { fromCognitoIdentityPool } from "@aws-sdk/credential-provider-cognito-identity";
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
```

```
const REGION = "REGION";
const IDENTITY_POOL_ID = "IDENTITY_POOL_ID"; // An Amazon Cognito Identity Pool ID.

// Create an Amazon DynamoDB service client object.
const dynamoClient = new DynamoDBClient({
  region: REGION,
  credentials: fromCognitoIdentityPool({
    client: new CognitoIdentityClient({ region: REGION }),
    identityPoolId: IDENTITY_POOL_ID,
  }),
});

export { dynamoClient };
```

This code is available [here on GitHub](#).

Next, create a `dynamoAppHelperFiles` folder in your project folder, create a file `update-table.js` in it, and copy the content [here on GitHub](#) into it.

```
// Import required AWS SDK clients and commands for Node.js
import { PutItemCommand } from "@aws-sdk/client-dynamodb";
import { dynamoClient } from "../libs/dynamoClient.js";

// Set the parameters
export const params = {
  TableName: "Items",
  Item: {
    id: { N: "1" },
    title: { S: "aTitle" },
    name: { S: "aName" },
    body: { S: "aBody" },
  },
};

export const run = async () => {
  try {
    const data = await dynamoClient.send(new PutItemCommand(params));
    console.log("success");
    console.log(data);
  } catch (err) {
    console.error(err);
  }
}
```



```
};  
run();
```

Run the following command from the command line.

```
node update-table.js
```

This code is available [here on GitHub](#).

Create a front-end page for the app

Here you create the front-end HTML browser page for the app.

Create a `DynamoDBApp` directory, create a file named `index.html`, and copy in code from [here on GitHub](#). The `script` element adds the `main.js` file, which contains all the required JavaScript for the example. You will create the `main.js` file later in this tutorial. The remaining code in `index.html` creates the browser page that captures the data that users input.

This example code can be found [here on GitHub](#).

Create the browser script

First, create the service client objects required for the example. Create a `libs` directory, create `snsClient.js`, and paste the code below into it. Replace *REGION* and *IDENTITY_POOL_ID* in each.

Note

Use the ID of the Amazon Cognito identity pool you created in [Create the AWS resources](#).

```
import { CognitoIdentityClient } from "@aws-sdk/client-cognito-identity";  
import { fromCognitoIdentityPool } from "@aws-sdk/credential-provider-cognito-identity";  
import { SNSClient } from "@aws-sdk/client-sns";  
  
const REGION = "REGION";  
const IDENTITY_POOL_ID = "IDENTITY_POOL_ID"; // An Amazon Cognito Identity Pool ID.  
  
// Create an Amazon Comprehend service client object.
```

```
const snsClient = new SNSClient({
  region: REGION,
  credentials: fromCognitoIdentityPool({
    client: new CognitoIdentityClient({ region: REGION }),
    identityPoolId: IDENTITY_POOL_ID,
  }),
});

export { snsClient };
```

This code is available [here on GitHub](#).

To create the browser script for this example, in a folder named `DynamoDBApp`, create a Node.js module with the file name `add_data.js` and paste the code below into it. The `submitData` function submits data to a DynamoDB table, and sends an SMS text to the app administrator using Amazon SNS.

In the `submitData` function, declare variables for the target phone number, the values entered on the app interface, and for the name of the Amazon S3 bucket. Next, create a parameters object for adding an item to the table. If none of the values is empty, `submitData` adds the item to the table, and sends the message. Remember to make the function available to the browser, with `window.submitData = submitData`.

```
// Import required AWS SDK clients and commands for Node.js
import { PutItemCommand } from "@aws-sdk/client-dynamodb";
import { PublishCommand } from "@aws-sdk/client-sns";
import { snsClient } from "../libs/snsClient.js";
import { dynamoClient } from "../libs/dynamoClient.js";

export const submitData = async () => {
  //Set the parameters
  // Capture the values entered in each field in the browser (by id).
  const id = document.getElementById("id").value;
  const title = document.getElementById("title").value;
  const name = document.getElementById("name").value;
  const body = document.getElementById("body").value;
  //Set the table name.
  const tableName = "Items";

  //Set the parameters for the table
  const params = {
    TableName: tableName,
```

```
// Define the attributes and values of the item to be added. Adding ' + "" '
converts a value to
// a string.
Item: {
  id: { N: id + "" },
  title: { S: title + "" },
  name: { S: name + "" },
  body: { S: body + "" },
},
};
// Check that all the fields are completed.
if (id !== "" && title !== "" && name !== "" && body !== "") {
  try {
    //Upload the item to the table
    await dynamoClient.send(new PutItemCommand(params));
    alert("Data added to table.");
    try {
      // Create the message parameters object.
      const messageParams = {
        Message: "A new item with ID value was added to the DynamoDB",
        PhoneNumber: "PHONE_NUMBER", //PHONE_NUMBER, in the E.164 phone number
        structure.
        // For example, a standard local formatted number, such as (415) 555-2671,
        is +14155552671 in E.164
        // format, where '1' is the country code.
      };
      // Send the SNS message
      const data = await snsClient.send(new PublishCommand(messageParams));
      console.log(
        "Success, message published. MessageID is " + data.MessageId,
      );
    } catch (err) {
      // Display error message if error is not sent
      console.error(err, err.stack);
    }
  } catch (err) {
    // Display error message if item is not added to table
    console.error(
      "An error occurred. Check the console for further information",
      err,
    );
  }
  // Display alert if all fields are not completed.
} else {
```

```
    alert("Enter data in each field.");
  }
};
// Expose the function to the browser
window.submitData = submitData;
```

This example code can be found [here on GitHub](#).

Finally, run the following at the command prompt to bundle the JavaScript for this example in a file named `main.js`:

```
webpack add_data.js --mode development --target web --devtool false -o main.js
```

Note

For information about installing webpack, see [Bundle applications with webpack](#).

To run the app, open `index.html` on your browser.

Delete the resources

As stated at the beginning of this tutorial, be sure to terminate all of the resources you create while going through this tutorial to ensure that you're not charged. You can do this by deleting the AWS CloudFormation stack you created in the [Create the AWS resources](#) topic of this tutorial, as follows:

1. Open the [AWS CloudFormation in the AWS management console](#).
2. Open the **Stacks** page, and select the stack.
3. Choose **Delete**.

For more AWS cross-service examples, see [AWS SDK for JavaScript cross-service examples](#).

Invoking Lambda with API Gateway

You can invoke a Lambda function by using Amazon API Gateway, which is an AWS service for creating, publishing, maintaining, monitoring, and securing REST, HTTP, and WebSocket APIs at scale. API developers can create APIs that access AWS or other web services, as well as data stored in the AWS Cloud. As an API Gateway developer, you can create APIs for use in your own client applications. For more information, see [What is Amazon API Gateway](#).

AWS Lambda is a compute service that enables you to run code without provisioning or managing servers. You can create Lambda functions in various programming languages. For more information about AWS Lambda, see [What is AWS Lambda](#).

In this example, you create a Lambda function by using the Lambda JavaScript runtime API. This example invokes different AWS services to perform a specific use case. For example, assume that an organization sends a mobile text message to its employees that congratulates them at the one year anniversary date, as shown in this illustration.



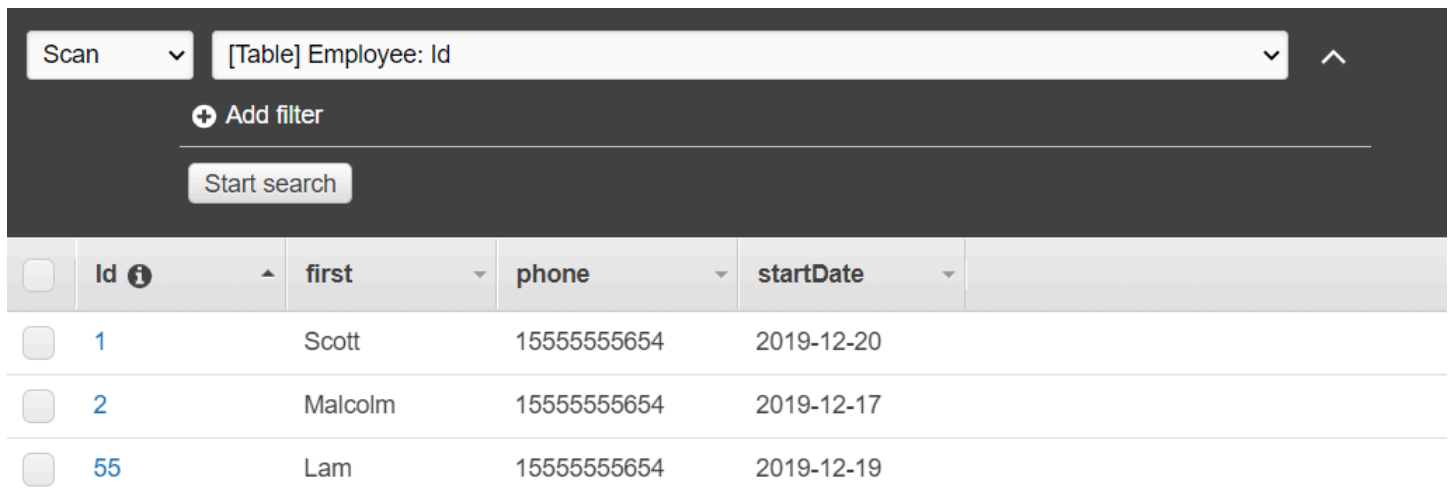
The example should take about 20 minutes to complete.

This example shows you how to use JavaScript logic to create a solution that performs this use case. For example, you'll learn how to read a database to determine which employees have reached the one year anniversary date, how to process the data, and send out a text message all by using a Lambda function. Then you'll learn how to use API Gateway to invoke this AWS Lambda function by using a Rest endpoint. For example, you can invoke the Lambda function by using this curl command:

```
curl -XGET "https://xxxxqjko1o3.execute-api.us-east-1.amazonaws.com/cronstage/employee"
```

This AWS tutorial uses an Amazon DynamoDB table named Employee that contains these fields.

- **id** - the primary key for the table.
- **firstName** - employee's first name.
- **phone** - employee's phone number.
- **startDate** - employee's start date.



The screenshot shows a web application interface. At the top, there is a search bar with a dropdown menu set to 'Scan' and a search input field containing '[Table] Employee: Id'. Below the search bar is a '+ Add filter' button and a 'Start search' button. Below the search bar is a table with the following columns: 'Id', 'first', 'phone', and 'startDate'. The table contains three rows of data:

	Id	first	phone	startDate
<input type="checkbox"/>	1	Scott	15555555654	2019-12-20
<input type="checkbox"/>	2	Malcolm	15555555654	2019-12-17
<input type="checkbox"/>	55	Lam	15555555654	2019-12-19

Important

Cost to complete: The AWS services included in this document are included in the AWS Free Tier. However, be sure to terminate all of the resources after you have completed this example to ensure that you are not charged.

To build the app:

1. [Complete prerequisites](#)
2. [Create the AWS resources](#)
3. [Prepare the browser script](#)
4. [Create and upload Lambda function](#)
5. [Deploy the Lambda function](#)
6. [Run the app](#)
7. [Delete the resources](#)

Prerequisite tasks

To set up and run this example, you must first complete these tasks:

- Set up the project environment to run these Node TypeScript examples, and install the required AWS SDK for JavaScript and third-party modules. Follow the instructions on [GitHub](#).

- Create a shared configurations file with your user credentials. For more information about providing a shared credentials file, see [Shared config and credentials files](#) in the *AWS SDKs and Tools Reference Guide*.

Create the AWS resources

This tutorial requires the following resources:

- An Amazon DynamoDB table named `Employee` with a key named `Id` and the fields shown in the previous illustration. Make sure you enter the correct data, including a valid mobile phone that you want to test this use case with. For more information, see [Create a Table](#).
- An IAM role with attached permissions to execute Lambda functions.
- An Amazon S3 bucket to host Lambda function.

You can create these resources manually, but we recommend provisioning these resources using the AWS CloudFormation as described in this tutorial.

Create the AWS resources using AWS CloudFormation

AWS CloudFormation enables you to create and provision AWS infrastructure deployments predictably and repeatedly. For more information about AWS CloudFormation, see the [AWS CloudFormation User Guide](#).

To create the AWS CloudFormation stack using the AWS CLI:

1. Install and configure the AWS CLI following the instructions in the [AWS CLI User Guide](#).
2. Create a file named `setup.yaml` in the root directory of your project folder, and copy the content [here on GitHub](#) into it.

Note

The AWS CloudFormation template was generated using the AWS CDK available [here on GitHub](#). For more information about the AWS CDK, see the [AWS Cloud Development Kit \(AWS CDK\) Developer Guide](#).

3. Run the following command from the command line, replacing `STACK_NAME` with a unique name for the stack.

⚠ Important

The stack name must be unique within an AWS Region and AWS account. You can specify up to 128 characters, and numbers and hyphens are allowed.

```
aws cloudformation create-stack --stack-name STACK_NAME --template-body file://
setup.yaml --capabilities CAPABILITY_IAM
```

For more information on the `create-stack` command parameters, see the [AWS CLI Command Reference guide](#), and the [AWS CloudFormation User Guide](#).

4. Next, populate the table by following the procedure [Populating the table](#).

Populating the table

To populate the table, first create a directory named `libs`, and in it create a file named `dynamoClient.js`, and paste the content below into it.

```
const { DynamoDBClient } = require ( "@aws-sdk/client-dynamodb" );
// Set the AWS Region.
const REGION = "REGION"; // e.g. "us-east-1"
// Create an Amazon Lambda service client object.
const dynamoClient = new DynamoDBClient({region:REGION});
module.exports = { dynamoClient };
```

This code is available [here on GitHub](#).

Next, create a file named `populate-table.js` in the root directory of your project folder, and copy the content [here on GitHub](#) into it. For one of the items, replace the value for the `phone` property with a valid mobile phone number in the E.164 format, and the value for the `startDate` with today's date.

Run the following command from the command line.

```
node populate-table.js
```



```
const { BatchWriteItemCommand } = require ( "aws-sdk/client-dynamodb" );
const {dynamoClient} = require ( "./libs/dynamoClient" );

// Set the parameters.
export const params = {
  RequestItems: {
    Employees: [
      {
        PutRequest: {
          Item: {
            id: { N: "1" },
            firstName: { S: "Bob" },
            phone: { N: "1555555555555654" },
            startDate: { S: "2019-12-20" },
          },
        },
      },
      {
        PutRequest: {
          Item: {
            id: { N: "2" },
            firstName: { S: "Xing" },
            phone: { N: "1555555555555653" },
            startDate: { S: "2019-12-17" },
          },
        },
      },
      {
        PutRequest: {
          Item: {
            id: { N: "55" },
            firstName: { S: "Harriette" },
            phone: { N: "1555555555555652" },
            startDate: { S: "2019-12-19" },
          },
        },
      },
    ],
  },
};

export const run = async () => {
  try {
    const data = await dbclient.send(new BatchWriteItemCommand(params));
  }
}
```

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

This code is available [here on GitHub](#).

Creating the AWS Lambda function

Configuring the SDK

In the `libs` directory, create files named `snsClient.js` and `lambdaClient.js`, and paste the content below into these files, respectively.

```
const { SNSClient } = require ( "@aws-sdk/client-sns" );
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon SNS service client object.
const snsClient = new SNSClient({ region: REGION });
module.exports = { snsClient };
```

Replace *REGION* with the AWS Region. This code is available [here on GitHub](#).

```
const { LambdaClient } = require ( "@aws-sdk/client-lambda" );
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon Lambda service client object.
const lambdaClient = new LambdaClient({ region: REGION });
module.exports = { lambdaClient };
```

Replace *REGION* with the AWS Region. This code is available [here on GitHub](#).

First, import the required AWS SDK for JavaScript (v3) modules and commands. Then calculate today's date and assign it to a parameter. Third, create the parameters for the ScanCommand. Replace *TABLE_NAME* with the name of the table you created in the [Create the AWS resources](#) section of this example.

The following code snippet shows this step. (See [Bundling the Lambda function](#) for the full example.)

```
"use strict";
const { ScanCommand } = require("@aws-sdk/client-dynamodb");
const { PublishCommand } = require("@aws-sdk/client-sns");
const {snsClient} = require ( "./libs/snsClient" );
const {dynamoClient} = require ( "./libs/dynamoClient" );

// Get today's date.
const today = new Date();
const dd = String(today.getDate()).padStart(2, "0");
const mm = String(today.getMonth() + 1).padStart(2, "0"); //January is 0!
const yyyy = today.getFullYear();
const date = yyyy + "-" + mm + "-" + dd;

// Set the parameters for the ScanCommand method.
const params = {
  // Specify which items in the results are returned.
  FilterExpression: "startDate = :topic",
  // Define the expression attribute value, which are substitutes for the values you
  want to compare.
  ExpressionAttributeValues: {
    ":topic": { S: date },
  },
  // Set the projection expression, which are the attributes that you want.
  ProjectionExpression: "firstName, phone",
  TableName: "Employees",
};
```

Scanning the DynamoDB table

First, create an `async/await` function called `sendText` to publish a text message using the Amazon SNS `PublishCommand`. Then, add a `try` block pattern that scans the DynamoDB table for employees with their work anniversary today, and then calls the `sendText` function to send these employees a text message. If an error occurs the `catch` block is called.

The following code snippet shows this step. (See [Bundling the Lambda function](#) for the full example.)

```
// Helper function to send message using Amazon SNS.
exports.handler = async () => {
```

```
// Helper function to send message using Amazon SNS.
async function sendText(textParams) {
  try {
    await snsClient.send(new PublishCommand(textParams));
    console.log("Message sent");
  } catch (err) {
    console.log("Error, message not sent ", err);
  }
}
try {
  // Scan the table to identify employees with work anniversary today.
  const data = await dynamoClient.send(new ScanCommand(params));
  data.Items.forEach(function (element) {
    const textParams = {
      PhoneNumber: element.phone.N,
      Message:
        "Hi " +
        element.firstName.S +
        "; congratulations on your work anniversary!",
    };
    // Send message using Amazon SNS.
    sendText(textParams);
  });
} catch (err) {
  console.log("Error, could not scan table ", err);
}
};
```

Bundling the Lambda function

This topic describes how to bundle the `mylambdafunction.ts` and the required AWS SDK for JavaScript modules for this example into a bundled file called `index.js`.

1. If you haven't already, follow the [Prerequisite tasks](#) for this example to install webpack.

Note

For information about *webpack*, see [Bundle applications with webpack](#).

2. Run the following in the command line to bundle the JavaScript for this example into a file called `<index.js>`:

```
webpack mylambdafunction.ts --mode development --target node --devtool false --
output-library-target umd -o index.js
```

Important

Notice the output is named `index.js`. This is because Lambda functions must have an `index.js` handler to work.

3. Compress the bundled output file, `index.js`, into a ZIP file named `mylambdafunction.zip`.
4. Upload `mylambdafunction.zip` to the Amazon S3 bucket you created in the [Create the AWS resources](#) topic of this tutorial.

Deploy the Lambda function

In the root of your project, create a `lambda-function-setup.ts` file, and paste the content below into it.

Replace `BUCKET_NAME` with the name of the Amazon S3 bucket you uploaded the ZIP version of your Lambda function to. Replace `ZIP_FILE_NAME` with the name of the ZIP version of your Lambda function. Replace `ROLE` with the Amazon Resource Number (ARN) of the IAM role you created in the [Create the AWS resources](#) topic of this tutorial. Replace `LAMBDA_FUNCTION_NAME` with a name for the Lambda function.

```
// Load the required Lambda client and commands.
const {
  CreateFunctionCommand
} = require ( "@aws-sdk/client-lambda" );
const { lambdaClient } = require ( "../libs/lambdaClient.js" );

// Set the parameters.
const params = {
  Code: {
    S3Bucket: "BUCKET_NAME", // BUCKET_NAME
    S3Key: "ZIP_FILE_NAME", // ZIP_FILE_NAME
  },
  FunctionName: "LAMBDA_FUNCTION_NAME",
  Handler: "index.handler",
  Role: "IAM_ROLE_ARN", // IAM_ROLE_ARN; e.g., arn:aws:iam::650138640062:role/v3-
  lambda-tutorial-lambda-role
```

```
Runtime: "nodejs12.x",
Description:
  "Scans a DynamoDB table of employee details and using Amazon Simple Notification
  Services (Amazon SNS) to " +
  "send employees an email on each anniversary of their start-date.",
};

const run = async () => {
  try {
    const data = await lambdaClient.send(new CreateFunctionCommand(params));
    console.log("Success", data); // successful response
  } catch (err) {
    console.log("Error", err); // an error occurred
  }
};
run();
```

Enter the following at the command line to deploy the Lambda function.

```
node lambda-function-setup.ts
```

This code example is available [here on GitHub](#).

Configure API Gateway to invoke the Lambda function

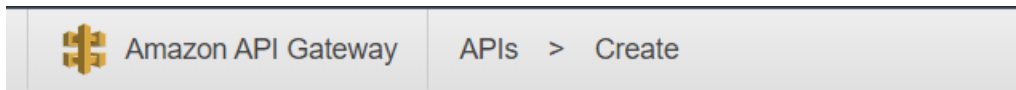
To build the app:

1. [Create the rest API](#)
2. [Test the API Gateway method](#)
3. [Deploy the API Gateway method](#)

Create the rest API

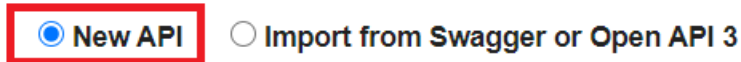
You can use the API Gateway console to create a rest endpoint for the Lambda function. Once done, you are able to invoke the Lambda function using a restful call.

1. Sign in to the [Amazon API Gateway console](#).
2. Under Rest API, choose **Build**.
3. Select **New API**.



Create new API

In Amazon API Gateway, a REST API refers to a collection of resources and meth



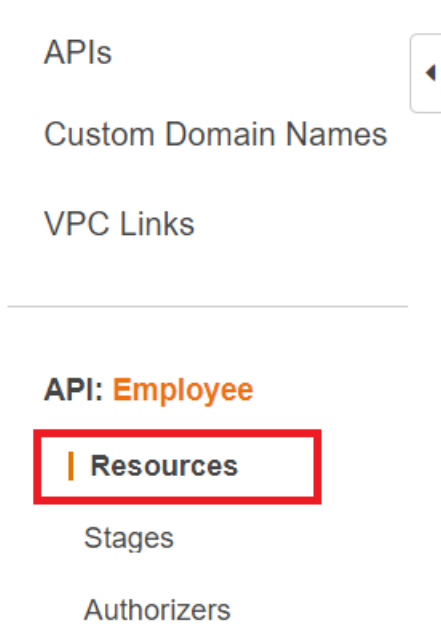
4. Specify **Employee** as the API name and provide a description.

Settings

Choose a friendly name and description for your API.


API name*	<input type="text" value="Employee"/>
Description	<input type="text" value="This invokes a Lambda function"/>
Endpoint Type	<input type="text" value="Regional"/> ⓘ

5. Choose **Create API**.
6. Choose **Resources** under the **Employee** section.



7. In the name field, specify **employees**.

8. Choose **Create Resources**.
9. From the **Actions** dropdown, choose **Create Resources**.

Use this page to create a new child resource for your resource. 

Configure as [proxy resource](#) 

Resource Name*

Resource Path*

You can add path parameters using brackets. For example, the resource path `{username}` represents a path parameter called 'username'. Configuring `{proxy+}` as a proxy resource catches all requests to its sub-resources. For example, it works for a GET request to `/foo`. To handle requests to `/`, add a new ANY method on the `/` resource.

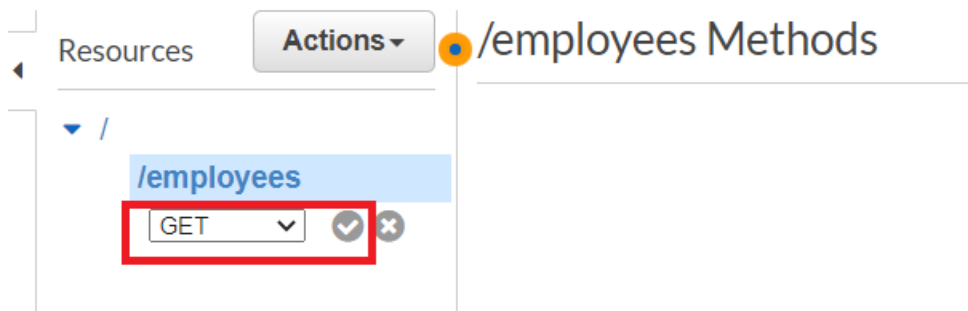
Enable API Gateway CORS 

* Required

Cancel

Create Resource

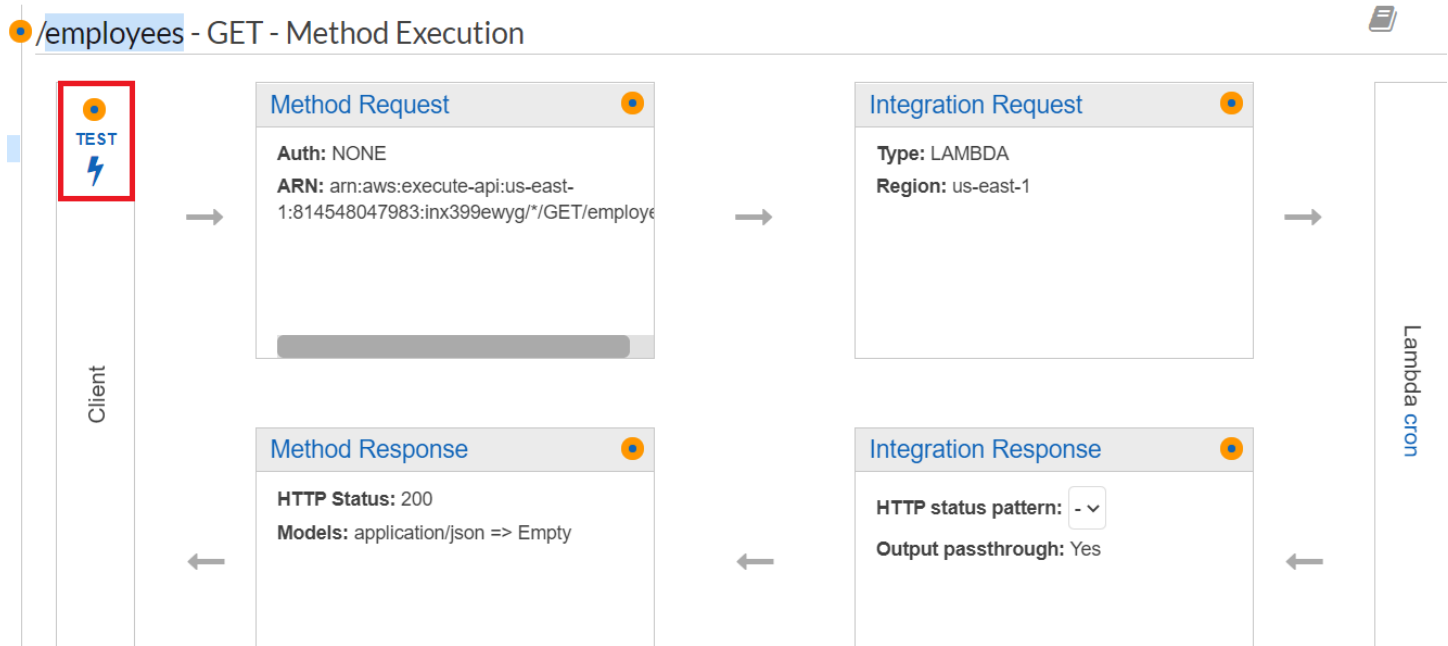
10. Choose `/employees`, select **Create Method** from the **Actions**, then select **GET** from the dropdown menu below `/employees`. Choose the checkmark icon.



11. Choose **Lambda function** and enter `mylambdafunction` as the Lambda function name. Choose **Save**.

Test the API Gateway method

At this point in the tutorial, you can test the API Gateway method that invokes the `mylambdafunction` Lambda function. To test the method, choose **Test**, as shown in the following illustration.

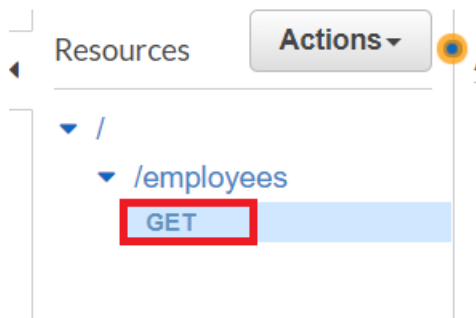


Once the Lambda function is invoked, you can view the log file to see a successful message.

Deploy the API Gateway method

After the test is successful, you can deploy the method from the [Amazon API Gateway console](#).

1. Choose **Get**.



2. From the **Actions** dropdown, select **Deploy API**.

Deploy API

Choose a stage where your API will be deployed. For example, a test version of your API could be deployed to a stage named beta.

Deployment stage	<input type="text" value="[New Stage]"/>
Stage name*	<input type="text" value="lambdastage"/>
Stage description	<input type="text"/>
Deployment description	<input type="text"/>

[Cancel](#) [Deploy](#)

3. Fill in the **Deploy API** form and choose **Deploy**.

Deploy API

Choose a stage where your API will be deployed. For example, a test version of your API could be deployed to a stage named beta.

Deployment stage	<input type="text" value="[New Stage]"/>
Stage name*	<input type="text" value="lambdastage"/>
Stage description	<input type="text"/>
Deployment description	<input type="text"/>

[Cancel](#) [Deploy](#)

4. Choose **Save Changes**.
5. Choose **Get** again and notice that the URL changes. This is the invocation URL that you can use to invoke the Lambda function.

The screenshot shows the AWS API Gateway console interface. On the left, a navigation pane shows a tree structure: 'lambdastage' > '/' > '/employees', with the 'GET' method selected and highlighted in a red box. A 'Create' button is visible at the top of the pane. The main content area is titled 'lambdastage - GET - /employees'. A light blue banner at the top of the main area contains the text 'Invoke URL: https://[redacted].execute-api.us-east-1.amazonaws.com/lambdastage/employees', which is also highlighted in a red box. Below the banner, there is a text prompt: 'Use this page to override the lambdastage stage settings for the GET to /employees method.' Underneath, there are two radio button options for 'Settings': 'Inherit from stage' (which is selected) and 'Override for this method'.

Delete the resources

Congratulations! You have invoked a Lambda function through Amazon API Gateway using the AWS SDK for JavaScript. As stated at the beginning of this tutorial, be sure to terminate all of the resources you create while going through this tutorial to ensure that you're not charged. You can do this by deleting the AWS CloudFormation stack you created in the [Create the AWS resources](#) topic of this tutorial, as follows:

1. Open the [AWS CloudFormation in the AWS management console](#).
2. Open the **Stacks** page, and select the stack.
3. Choose **Delete**.

Creating scheduled events to execute AWS Lambda functions

You can create a scheduled event that invokes an AWS Lambda function by using an Amazon CloudWatch Event. You can configure a CloudWatch Event to use a cron expression to schedule when a Lambda function is invoked. For example, you can schedule a CloudWatch Event to invoke an Lambda function every weekday.

AWS Lambda is a compute service that enables you to run code without provisioning or managing servers. You can create Lambda functions in various programming languages. For more information about AWS Lambda, see [What is AWS Lambda](#).

In this tutorial, you create a Lambda function by using the Lambda JavaScript runtime API. This example invokes different AWS services to perform a specific use case. For example, assume that an organization sends a mobile text message to its employees that congratulates them at the one year anniversary date, as shown in this illustration.



The tutorial should take about 20 minutes to complete.

This tutorial shows you how to use JavaScript logic to create a solution that performs this use case. For example, you'll learn how to read a database to determine which employees have reached the one year anniversary date, how to process the data, and send out a text message all by using a Lambda function. Then you'll learn how to use a cron expression to invoke the Lambda function every weekday.

This AWS tutorial uses an Amazon DynamoDB table named Employee that contains these fields.

- **id** - the primary key for the table.
- **firstName** - employee's first name.
- **phone** - employee's phone number.
- **startDate** - employee's start date.

<input type="checkbox"/>	Id <i>i</i>	first	phone	startDate
<input type="checkbox"/>	1	Scott	15555555654	2019-12-20
<input type="checkbox"/>	2	Malcolm	15555555654	2019-12-17
<input type="checkbox"/>	55	Lam	15555555654	2019-12-19

Important

Cost to complete: The AWS services included in this document are included in the AWS Free Tier. However, be sure to terminate all of the resources after you have completed this tutorial to ensure that you are not charged.

To build the app:

1. [Complete prerequisites](#)
2. [Create the AWS resources](#)
3. [Prepare the browser script](#)
4. [Create and upload Lambda function](#)
5. [Deploy the Lambda function](#)
6. [Run the app](#)
7. [Delete the resources](#)

Prerequisite tasks

To set up and run this example, you must first complete these tasks:

- Set up the project environment to run these Node.js TypeScript examples, and install the required AWS SDK for JavaScript and third-party modules. Follow the instructions on [GitHub](#).
- Create a shared configurations file with your user credentials. For more information about providing a shared credentials file, see [Shared config and credentials files](#) in the *AWS SDKs and Tools Reference Guide*.

Create the AWS resources

This tutorial requires the following resources.

- An Amazon DynamoDB table named **Employee** with a key named **Id** and the fields shown in the previous illustration. Make sure you enter the correct data, including a valid mobile phone that you want to test this use case with. For more information, see [Create a Table](#).
- An IAM role with attached permissions to execute Lambda functions.
- An Amazon S3 bucket to host Lambda function.

You can create these resources manually, but we recommend provisioning these resources using the AWS CloudFormation as described in this tutorial.

Create the AWS resources using AWS CloudFormation

AWS CloudFormation enables you to create and provision AWS infrastructure deployments predictably and repeatedly. For more information about AWS CloudFormation, see the [AWS CloudFormation User Guide](#).

To create the AWS CloudFormation stack using the AWS CLI:

1. Install and configure the AWS CLI following the instructions in the [AWS CLI User Guide](#).
2. Create a file named `setup.yaml` in the root directory of your project folder, and copy the content [here on GitHub](#) into it.

Note

The AWS CloudFormation template was generated using the AWS CDK available [here on GitHub](#). For more information about the AWS CDK, see the [AWS Cloud Development Kit \(AWS CDK\) Developer Guide](#).

3. Run the following command from the command line, replacing `STACK_NAME` with a unique name for the stack.

Important

The stack name must be unique within an AWS Region and AWS account. You can specify up to 128 characters, and numbers and hyphens are allowed.

```
aws cloudformation create-stack --stack-name STACK_NAME --template-body file://  
setup.yaml --capabilities CAPABILITY_IAM
```

For more information on the `create-stack` command parameters, see the [AWS CLI Command Reference guide](#), and the [AWS CloudFormation User Guide](#).

View a list of the resources in the console by opening the stack on the AWS CloudFormation dashboard, and choosing the **Resources** tab. You require these for the tutorial.

4. When the stack is created, use the AWS SDK for JavaScript to populate the DynamoDB table, as described in [Populate the DynamoDB table](#).

Populate the DynamoDB table

To populate the table, first create a directory named `libs`, and in it create a file named `dynamoClient.js`, and paste the content below into it.

```
const { DynamoDBClient } = require( "@aws-sdk/client-dynamodb" );
// Set the AWS Region.
const REGION = "REGION"; // e.g. "us-east-1"
// Create an Amazon DynamoDB service client object.
const dynamoClient = new DynamoDBClient({region:REGION});
module.exports = { dynamoClient };
```

This code is available [here on GitHub](#).

Next, create a file named `populate-table.js` in the root directory of your project folder, and copy the content [here on GitHub](#) into it. For one of the items, replace the value for the `phone` property with a valid mobile phone number in the E.164 format, and the value for the `startDate` with today's date.

Run the following command from the command line.

```
node populate-table.js
```

```
const {
BatchWriteItemCommand } = require( "aws-sdk/client-dynamodb" );
const {dynamoClient} = require( "../libs/dynamoClient" );
// Set the parameters.
const params = {
  RequestItems: {
    Employees: [
      {
        PutRequest: {
          Item: {
            id: { N: "1" },
            firstName: { S: "Bob" },
            phone: { N: "155555555555654" },
```

```
        startDate: { S: "2019-12-20" },
      },
    },
  },
  {
    PutRequest: {
      Item: {
        id: { N: "2" },
        firstName: { S: "Xing" },
        phone: { N: "155555555555653" },
        startDate: { S: "2019-12-17" },
      },
    },
  },
  {
    PutRequest: {
      Item: {
        id: { N: "55" },
        firstName: { S: "Harriette" },
        phone: { N: "155555555555652" },
        startDate: { S: "2019-12-19" },
      },
    },
  },
],
},
};

export const run = async () => {
  try {
    const data = await dbclient.send(new BatchWriteItemCommand(params));
    console.log("Success", data);
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

This code is available [here on GitHub](#).

Creating the AWS Lambda function

Configuring the SDK

First import the required AWS SDK for JavaScript (v3) modules and commands: `DynamoDBClient` and the `DynamoDB ScanCommand`, and `SNSClient` and the `Amazon SNS PublishCommand` command. Replace `REGION` with the AWS Region. Then calculate today's date and assign it to a parameter. Then create the parameters for the `ScanCommand`. Replace `TABLE_NAME` with the name of the table you created in the [Create the AWS resources](#) section of this example.

The following code snippet shows this step. (See [Bundling the Lambda function](#) for the full example.)

```
"use strict";
// Load the required clients and commands.
const { DynamoDBClient, ScanCommand } = require("@aws-sdk/client-dynamodb");
const { SNSClient, PublishCommand } = require("@aws-sdk/client-sns");

//Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"

// Get today's date.
const today = new Date();
const dd = String(today.getDate()).padStart(2, "0");
const mm = String(today.getMonth() + 1).padStart(2, "0"); //January is 0!
const yyyy = today.getFullYear();
const date = yyyy + "-" + mm + "-" + dd;

// Set the parameters for the ScanCommand method.
const params = {
  // Specify which items in the results are returned.
  FilterExpression: "startDate = :topic",
  // Define the expression attribute value, which are substitutes for the values you
  want to compare.
  ExpressionAttributeValues: {
    ":topic": { S: date },
  },
  // Set the projection expression, which the the attributes that you want.
  ProjectionExpression: "firstName, phone",
  TableName: "TABLE_NAME",
};
```

Scanning the DynamoDB table

First create an `async/await` function called `sendText` to publish a text message using the Amazon SNS `PublishCommand`. Then, add a `try` block pattern that scans the DynamoDB table for employees with their work anniversary today, and then calls the `sendText` function to send these employees a text message. If an error occurs the `catch` block is called.

The following code snippet shows this step. (See [Bundling the Lambda function](#) for the full example.)

```
exports.handler = async (event, context, callback) => {
  // Helper function to send message using Amazon SNS.
  async function sendText(textParams) {
    try {
      const data = await snsclient.send(new PublishCommand(textParams));
      console.log("Message sent");
    } catch (err) {
      console.log("Error, message not sent ", err);
    }
  }
  try {
    // Scan the table to check identify employees with work anniversary today.
    const data = await dbclient.send(new ScanCommand(params));
    data.Items.forEach(function (element, index, array) {
      const textParams = {
        PhoneNumber: element.phone.N,
        Message:
          "Hi " +
          element.firstName.S +
          "; congratulations on your work anniversary!",
      };
      // Send message using Amazon SNS.
      sendText(textParams);
    });
  } catch (err) {
    console.log("Error, could not scan table ", err);
  }
};
```

Bundling the Lambda function

This topic describes how to bundle the `mylambdafunction.js` and the required AWS SDK for JavaScript modules for this example into a bundled file called `index.js`.

1. If you haven't already, follow the [Prerequisite tasks](#) for this example to install webpack.

Note

For information about *webpack*, see [Bundle applications with webpack](#).

2. Run the the following in the command line to bundle the JavaScript for this example into a file called `<index.js>` :

```
webpack mylambdafunction.js --mode development --target node --devtool false --
output-library-target umd -o index.js
```

Important

Notice the output is named `index.js`. This is because Lambda functions must have an `index.js` handler to work.

3. Compress the bundled output file, `index.js`, into a ZIP file named `my-lambda-function.zip`.
4. Upload `mylambdafunction.zip` to the Amazon S3 bucket you created in the [Create the AWS resources](#) topic of this tutorial.

Here is the complete browser script code for `mylambdafunction.js`.

```
"use strict";
// Load the required clients and commands.
const { DynamoDBClient, ScanCommand } = require("@aws-sdk/client-dynamodb");
const { SNSClient, PublishCommand } = require("@aws-sdk/client-sns");

//Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"

// Get today's date.
const today = new Date();
const dd = String(today.getDate()).padStart(2, "0");
const mm = String(today.getMonth() + 1).padStart(2, "0"); //January is 0!
const yyyy = today.getFullYear();
const date = yyyy + "-" + mm + "-" + dd;
```

```
// Set the parameters for the ScanCommand method.
const params = {
  // Specify which items in the results are returned.
  FilterExpression: "startDate = :topic",
  // Define the expression attribute value, which are substitutes for the values you
  want to compare.
  ExpressionAttributeValues: {
    ":topic": { S: date },
  },
  // Set the projection expression, which the the attributes that you want.
  ProjectionExpression: "firstName, phone",
  TableName: "TABLE_NAME",
};

// Create the client service objects.
const dbclient = new DynamoDBClient({ region: REGION });
const snsclient = new SNSClient({ region: REGION });

exports.handler = async (event, context, callback) => {
  // Helper function to send message using Amazon SNS.
  async function sendText(textParams) {
    try {
      const data = await snsclient.send(new PublishCommand(textParams));
      console.log("Message sent");
    } catch (err) {
      console.log("Error, message not sent ", err);
    }
  }
  try {
    // Scan the table to check identify employees with work anniversary today.
    const data = await dbclient.send(new ScanCommand(params));
    data.Items.forEach(function (element, index, array) {
      const textParams = {
        PhoneNumber: element.phone.N,
        Message:
          "Hi " +
          element.firstName.S +
          "; congratulations on your work anniversary!",
      };
      // Send message using Amazon SNS.
      sendText(textParams);
    });
  } catch (err) {
    console.log("Error, could not scan table ", err);
  }
}
```

```
}  
};
```

Deploy the Lambda function

In the root of your project, create a `lambda-function-setup.js` file, and paste the content below into it.

Replace `BUCKET_NAME` with the name of the Amazon S3 bucket you uploaded the ZIP version of your Lambda function to. Replace `ZIP_FILE_NAME` with the name of the ZIP version of your Lambda function. Replace `IAM_ROLE_ARN` with the Amazon Resource Number (ARN) of the IAM role you created in the [Create the AWS resources](#) topic of this tutorial. Replace `LAMBDA_FUNCTION_NAME` with a name for the Lambda function.

```
// Load the required Lambda client and commands.  
const {  
  CreateFunctionCommand,  
} = require("@aws-sdk/client-lambda");  
const {  
  lambdaClient  
} = require("../libs/lambdaClient.js");  
  
// Instantiate an Lambda client service object.  
const lambda = new LambdaClient({ region: REGION });  
  
// Set the parameters.  
const params = {  
  Code: {  
    S3Bucket: "BUCKET_NAME", // BUCKET_NAME  
    S3Key: "ZIP_FILE_NAME", // ZIP_FILE_NAME  
  },  
  FunctionName: "LAMBDA_FUNCTION_NAME",  
  Handler: "index.handler",  
  Role: "IAM_ROLE_ARN", // IAM_ROLE_ARN; e.g., arn:aws:iam::650138640062:role/v3-lambda-tutorial-lambda-role  
  Runtime: "nodejs12.x",  
  Description:  
    "Scans a DynamoDB table of employee details and using Amazon Simple Notification  
    Services (Amazon SNS) to " +  
    "send employees an email the each anniversary of their start-date.",  
};
```

```
const run = async () => {
  try {
    const data = await lambda.send(new CreateFunctionCommand(params));
    console.log("Success", data); // successful response
  } catch (err) {
    console.log("Error", err); // an error occurred
  }
};
run();
```

Enter the following at the command line to deploy the Lambda function.

```
node lambda-function-setup.js
```

This code example is available [here on GitHub](#).

Configure CloudWatch to invoke the Lambda functions

To configure CloudWatch to invoke the Lambda functions:

1. Open the **Functions** page on the Lambda console.
2. Choose the Lambda function.
3. Under **Designer**, choose **Add trigger**.
4. Set the trigger type to **CloudWatch Events/EventBridge**.
5. For Rule, choose **Create a new rule**.
6. Fill in the Rule name and Rule description.
7. For rule type, select **Schedule expression**.
8. In the **Schedule expression** field, enter a cron expression. For example, **cron(0 12 ? * MON-FRI *)**.
9. Choose **Add**.

Note

For more information, see [Using Lambda with CloudWatch Events](#).

Delete the resources

Congratulations! You have invoked a Lambda function through Amazon CloudWatch scheduled events using the AWS SDK for JavaScript. As stated at the beginning of this tutorial, be sure to terminate all of the resources you create while going through this tutorial to ensure that you're not charged. You can do this by deleting the AWS CloudFormation stack you created in the [Create the AWS resources](#) topic of this tutorial, as follows:

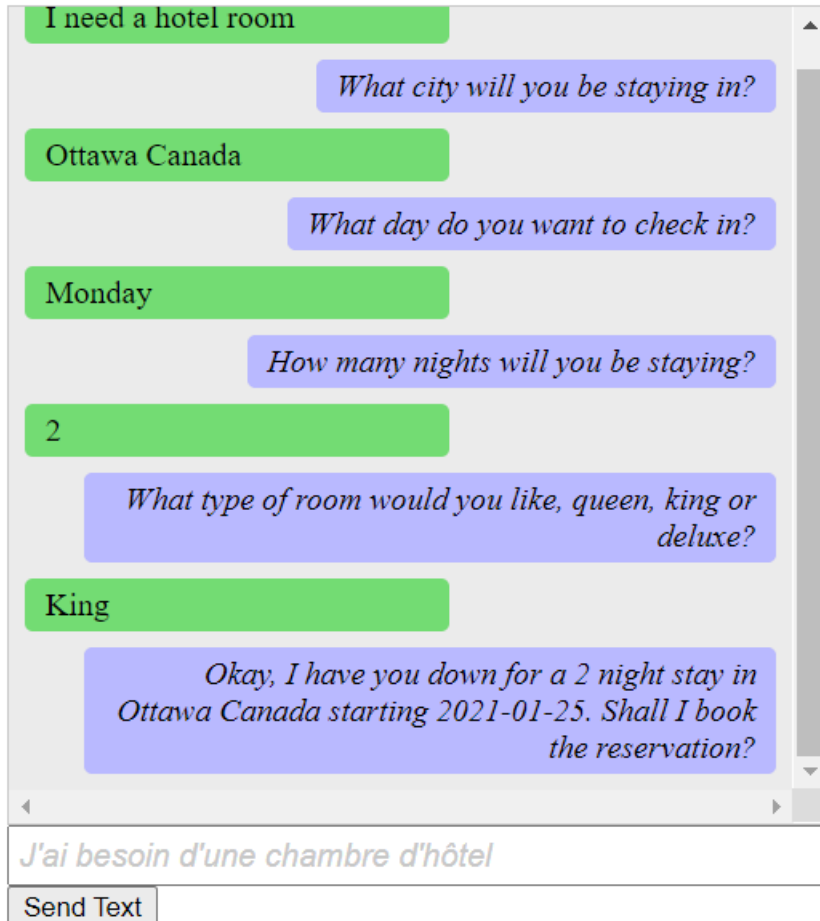
1. Open the [AWS CloudFormation console](#).
2. On the **Stacks** page, select the stack.
3. Choose **Delete**.

Building an Amazon Lex chatbot

You can create an Amazon Lex chatbot within a web application to engage your web site visitors. An Amazon Lex chatbot is functionality that performs on-line chat conversation with users without providing direct contact with a person. For example, the following illustration shows an Amazon Lex chatbot that engages a user about booking a hotel room.

Amazon Lex - BookTrip

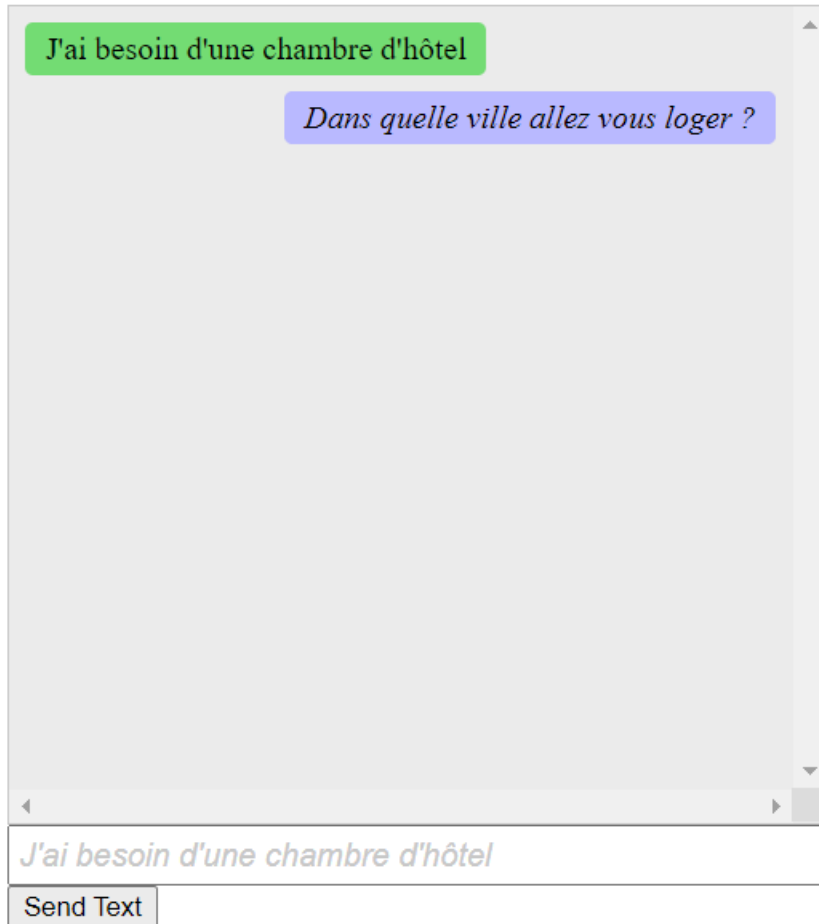
This multiple language chatbot shows you how easy it is to incorporate [Amazon Lex](#) into your web apps. Try it out.



The Amazon Lex chatbot created in this AWS tutorial is able to handle multiple languages. For example, a user who speaks French can enter French text and get back a response in French.

Amazon Lex - BookTrip

This little chatbot shows how easy it is to incorporate [Amazon Lex](#) into your web pages. Try it out.



Likewise, a user can communicate with the Amazon Lex chatbot in Italian.

Amazon Lex - BookTrip

This little chatbot shows how easy it is to incorporate [Amazon Lex](#) into your web pages. Try it out.



This AWS tutorial guides you through creating an Amazon Lex chatbot and integrating it into a Node.js web application. The AWS SDK for JavaScript (v3) is used to invoke these AWS services:

- Amazon Lex
- Amazon Comprehend
- Amazon Translate

Cost to complete: The AWS services included in this document are included in the [AWS Free Tier](#).

Note: Be sure to terminate all of the resources you create while going through this tutorial to ensure that you're not charged.

To build the app:

1. [Prerequisites](#)
2. [Provision resources](#)
3. [Create Amazon Lex chatbot](#)
4. [Create the HTML](#)

5. [Create the browser script](#)
6. [Next steps](#)

Prerequisites

To set up and run this example, you must first complete these tasks:

- Set up the project environment to run these Node TypeScript examples, and install the required AWS SDK for JavaScript and third-party modules. Follow the instructions on [GitHub](#).
- Create a shared configurations file with your user credentials. For more information about providing a shared credentials file, see [Shared config and credentials files](#) in the *AWS SDKs and Tools Reference Guide*.

Important

This example uses ECMAScript6 (ES6). This requires Node.js version 13.x or higher. To download and install the latest version of Node.js, see [Node.js downloads](#).

However, if you prefer to use CommonJS syntax, please refer to [JavaScript ES6/CommonJS syntax](#).

Create the AWS resources

This tutorial requires the following resources.

- An unauthenticated IAM role with attached permissions to:
 - Amazon Comprehend
 - Amazon Translate
 - Amazon Lex

You can create these resources manually, but we recommend provisioning these resources using AWS CloudFormation as described in this tutorial.

Create the AWS resources using AWS CloudFormation

AWS CloudFormation enables you to create and provision AWS infrastructure deployments predictably and repeatedly. For more information about AWS CloudFormation, see the [AWS CloudFormation User Guide](#).

To create the AWS CloudFormation stack using the AWS CLI:

1. Install and configure the AWS CLI following the instructions in the [AWS CLI User Guide](#).
2. Create a file named `setup.yaml` in the root directory of your project folder, and copy the content [here on GitHub](#) into it.

Note

The AWS CloudFormation template was generated using the AWS CDK available [here on GitHub](#). For more information about the AWS CDK, see the [AWS Cloud Development Kit \(AWS CDK\) Developer Guide](#).

3. Run the following command from the command line, replacing `STACK_NAME` with a unique name for the stack.

Important

The stack name must be unique within an AWS Region and AWS account. You can specify up to 128 characters, and numbers and hyphens are allowed.

```
aws cloudformation create-stack --stack-name STACK_NAME --template-body file://
setup.yaml --capabilities CAPABILITY_IAM
```

For more information on the `create-stack` command parameters, see the [AWS CLI Command Reference guide](#), and the [AWS CloudFormation User Guide](#).

To view the resources created, open the Amazon Lex console, choose the stack, and select the **Resources** tab.

Create an Amazon Lex bot

⚠ Important

Use V1 of the the Amazon Lex console to create the bot. This exmample does not work with bots created using V2.

The first step is to create an Amazon Lex chatbot by using the Amazon Web Services Management Console. In this example, the Amazon Lex **BookTrip** example is used. For more information, see [Book Trip](#).

- Sign in to the Amazon Web Services Management Console and open the Amazon Lex console at [Amazon Web Services Console](#).
- On the Bots page, choose **Create**.
- Choose **BookTrip** blueprint (leave the default bot name **BookTrip**).

Create your bot

Amazon Lex enables any developer to build conversational chatbots quickly and easily. With Amazon Lex, no deep learning expertise is necessary—you just specify the basic conversational flow directly from the console, and then Amazon Lex manages the dialogue and dynamically adjusts the response. To get started, you can choose one of the sample bots provided below or build a new custom bot from scratch.

The screenshot shows the 'Create your bot' interface in the Amazon Lex console. It features two main sections: 'CREATE YOUR OWN' and 'TRY A SAMPLE'. Under 'CREATE YOUR OWN', there is a 'Custom bot' button. Under 'TRY A SAMPLE', there are three buttons: 'BookTrip', 'OrderFlowers', and 'ScheduleAppointment'. The 'BookTrip' button is highlighted with a red border. Below these buttons is a 'Bot name' input field, which also contains the text 'BookTrip' and is highlighted with a red border.



- Fill in the default settings and choose **Create** (the console shows the **BookTrip** bot). On the Editor tab, review the details of the preconfigured intents.
- Test the bot in the test window. Start the test by typing *I want to book a hotel room.*

> Test bot (Latest)

✔ Ready. Build complete

I want to book a hotel room

What city will you be staying in?

Clear chat history

🎤 | Chat with your bot...

Inspect response

Dialog State: ElicitSlot

[Hide](#)

Summary Detail

Intent: BookHotel

- Choose **Publish** and specify an alias name (you will need this value when using the AWS SDK for JavaScript).

Note

You need to reference the **bot name** and the **bot alias** in your JavaScript code.

Create the HTML

Create a file named `index.html`. Copy and paste the code below in to `index.html`. This HTML references `main.js`. This is a bundled version of `index.js`, which includes the required AWS SDK for JavaScript modules. You'll create this file in [Create the HTML](#). `index.html` also references `style.css`, which adds the styles.

```
<!doctype html>
<head>
  <title>Amazon Lex - Sample Application (BookTrip)</title>
```

```
<link type="text/css" rel="stylesheet" href="style.css" />
</head>

<body>
  <h1 id="title">Amazon Lex - BookTrip</h1>
  <p id="intro">
    This multiple language chatbot shows you how easy it is to incorporate
    <a
      href="https://aws.amazon.com/lex/"
      title="Amazon Lex (product)"
      target="_new"
    >Amazon Lex</a>
    >
    into your web apps. Try it out.
  </p>
  <div id="conversation"></div>
  <input
    type="text"
    id="wisdom"
    size="80"
    value=""
    placeholder="J'ai besoin d'une chambre d'hôtel"
  />
  <br />
  <button onclick="createResponse()">Send Text</button>
  <script type="text/javascript" src="./main.js"></script>
</body>
```

This code is also available [here on GitHub](#).

Create the browser script

Create a file named `index.js`. Copy and paste the code below into `index.js`. Import the required AWS SDK for JavaScript modules and commands. Create clients for Amazon Lex, Amazon Comprehend, and Amazon Translate. Replace *REGION* with AWS Region, and *IDENTITY_POOL_ID* with the ID of the identity pool you created in the [Create the AWS resources](#). To retrieve this identity pool ID, open the identity pool in the Amazon Cognito console, choose **Edit identity pool**, and choose **Sample code** in the side menu. The identity pool ID is shown in red text in the console.

First, create a `libs` directory create the required service client objects by creating three files, `comprehendClient.js`, `lexClient.js`, and `translateClient.js`. Paste the appropriate code below into each, and replace *REGION* and *IDENTITY_POOL_ID* in each file.

Note

Use the ID of the Amazon Cognito identity pool you created in [Create the AWS resources using AWS CloudFormation](#).

```
import { CognitoIdentityClient } from "@aws-sdk/client-cognito-identity";
import { fromCognitoIdentityPool } from "@aws-sdk/credential-provider-cognito-identity";
import { ComprehendClient } from "@aws-sdk/client-comprehend";

const REGION = "REGION";
const IDENTITY_POOL_ID = "IDENTITY_POOL_ID"; // An Amazon Cognito Identity Pool ID.

// Create an Amazon Comprehend service client object.
const comprehendClient = new ComprehendClient({
  region: REGION,
  credentials: fromCognitoIdentityPool({
    client: new CognitoIdentityClient({ region: REGION }),
    identityPoolId: IDENTITY_POOL_ID,
  }),
});

export { comprehendClient };
```

```
import { CognitoIdentityClient } from "@aws-sdk/client-cognito-identity";
import { fromCognitoIdentityPool } from "@aws-sdk/credential-provider-cognito-identity";
import { LexRuntimeServiceClient } from "@aws-sdk/client-lex-runtime-service";

const REGION = "REGION";
const IDENTITY_POOL_ID = "IDENTITY_POOL_ID"; // An Amazon Cognito Identity Pool ID.

// Create an Amazon Lex service client object.
const lexClient = new LexRuntimeServiceClient({
  region: REGION,
  credentials: fromCognitoIdentityPool({
    client: new CognitoIdentityClient({ region: REGION }),
    identityPoolId: IDENTITY_POOL_ID,
  }),
});
```



```
export { lexClient };
```

```
import { CognitoIdentityClient } from "@aws-sdk/client-cognito-identity";
import { fromCognitoIdentityPool } from "@aws-sdk/credential-provider-cognito-identity";
import { TranslateClient } from "@aws-sdk/client-translate";

const REGION = "REGION";
const IDENTITY_POOL_ID = "IDENTITY_POOL_ID"; // An Amazon Cognito Identity Pool ID.

// Create an Amazon Translate service client object.
const translateClient = new TranslateClient({
  region: REGION,
  credentials: fromCognitoIdentityPool({
    client: new CognitoIdentityClient({ region: REGION }),
    identityPoolId: IDENTITY_POOL_ID,
  }),
});

export { translateClient };
```

This code is available [here on GitHub](#).

Next, create an `index.js` file, and paste the code below into it.

Replace `BOT_ALIAS` and `BOT_NAME` with the alias and name of your Amazon Lex bot respectively, and `USER_ID` with a user id. The `createResponse` asynchronous function does the following:

- Takes the text inputted by the user into the browser and uses Amazon Comprehend to determine its language code.
- Takes the language code and uses Amazon Translate to translate the text into English.
- Takes the translated text and uses Amazon Lex to generate a response.
- Posts the response to the browser page.

```
import { DetectDominantLanguageCommand } from "@aws-sdk/client-comprehend";
import { TranslateTextCommand } from "@aws-sdk/client-translate";
import { PostTextCommand } from "@aws-sdk/client-lex-runtime-service";
import { lexClient } from "../libs/lexClient.js";
import { translateClient } from "../libs/translateClient.js";
import { comprehendClient } from "../libs/comprehendClient.js";
```

```
var g_text = "";
// Set the focus to the input box.
document.getElementById("wisdom").focus();

function showRequest() {
  var conversationDiv = document.getElementById("conversation");
  var requestPara = document.createElement("P");
  requestPara.className = "userRequest";
  requestPara.appendChild(document.createTextNode(g_text));
  conversationDiv.appendChild(requestPara);
  conversationDiv.scrollTop = conversationDiv.scrollHeight;
}

function showResponse(lexResponse) {
  var conversationDiv = document.getElementById("conversation");
  var responsePara = document.createElement("P");
  responsePara.className = "lexResponse";

  var lexTextResponse = lexResponse;

  responsePara.appendChild(document.createTextNode(lexTextResponse));
  responsePara.appendChild(document.createElement("br"));
  conversationDiv.appendChild(responsePara);
  conversationDiv.scrollTop = conversationDiv.scrollHeight;
}

function handletext(text) {
  g_text = text;
  var xhr = new XMLHttpRequest();
  xhr.addEventListener("load", loadNewItems, false);
  xhr.open("POST", "../text", true); // A Spring MVC controller
  xhr.setRequestHeader("Content-type", "application/x-www-form-urlencoded"); //
  necessary
  xhr.send("text=" + text);
}

function loadNewItems() {
  showRequest();

  // Re-enable input.
  var wisdomText = document.getElementById("wisdom");
  wisdomText.value = "";
  wisdomText.locked = false;
}
```

```
}

// Respond to user's input.
const createResponse = async () => {
  // Confirm there is text to submit.
  var wisdomText = document.getElementById("wisdom");
  if (wisdomText && wisdomText.value && wisdomText.value.trim().length > 0) {
    // Disable input to show it is being sent.
    var wisdom = wisdomText.value.trim();
    wisdomText.value = "...";
    wisdomText.locked = true;
    handleText(wisdom);

    const comprehendParams = {
      Text: wisdom,
    };
    try {
      const data = await comprehendClient.send(
        new DetectDominantLanguageCommand(comprehendParams)
      );
      console.log(
        "Success. The language code is: ",
        data.Languages[0].LanguageCode
      );
      const translateParams = {
        SourceLanguageCode: data.Languages[0].LanguageCode,
        TargetLanguageCode: "en", // For example, "en" for English.
        Text: wisdom,
      };
      try {
        const data = await translateClient.send(
          new TranslateTextCommand(translateParams)
        );
        console.log("Success. Translated text: ", data.TranslatedText);
        const lexParams = {
          botName: "BookTrip",
          botAlias: "mynewalias",
          inputText: data.TranslatedText,
          userId: "chatbot", // For example, 'chatbot-demo'.
        };
      }
      try {
        const data = await lexClient.send(new PostTextCommand(lexParams));
        console.log("Success. Response is: ", data.message);
        var msg = data.message;
```

```
        showResponse(msg);
    } catch (err) {
        console.log("Error responding to message. ", err);
    }
} catch (err) {
    console.log("Error translating text. ", err);
}
} catch (err) {
    console.log("Error identifying language. ", err);
}
}
};
// Make the function available to the browser.
window.createResponse = createResponse;
```

This code is available [here on GitHub](#).

Now use webpack to bundle the `index.js` and AWS SDK for JavaScript modules into a single file, `main.js`.

1. If you haven't already, follow the [Prerequisites](#) for this example to install webpack.

Note

For information about *webpack*, see [Bundle applications with webpack](#).

2. Run the the following in the command line to bundle the JavaScript for this example into a file called `main.js`:

```
webpack index.js --mode development --target web --devtool false -o main.js
```

Next steps

Congratulations! You have created a Node.js application that uses Amazon Lex to create an interactive user experience. As stated at the beginning of this tutorial, be sure to terminate all of the resources you create while going through this tutorial to ensure that you're not charged. You can do this by deleting the AWS CloudFormation stack you created in the [Create the AWS resources](#) topic of this tutorial, as follows:

1. Open the [AWS CloudFormation console](#).

2. On the **Stacks** page, select the stack.
3. Choose **Delete**.

For more AWS cross-service examples, see [AWS SDK for JavaScript cross-service examples](#).

Creating an example messaging application

You can create an AWS application that sends and retrieves messages by using the AWS SDK for JavaScript and Amazon Simple Queue Service (Amazon SQS). Messages are stored in a first in, first out (FIFO) queue that ensures that the order of the messages is consistent. For example, the first message that's stored in the queue is the first message read from the queue.

Note

For more information about Amazon SQS, see [What is Amazon Simple Queue Service?](#)

In this tutorial, you create a Node.js application named AWS Messaging.

Cost to complete: The AWS services included in this document are included in the [AWS Free Tier](#).

Note: Be sure to terminate all of the resources you create while going through this tutorial to ensure that you're not charged.

To build the app:

1. [Prerequisites](#)
2. [Provision resources](#)
3. [Understand the workflow](#)
4. [Create the HTML](#)
5. [Create the browser script](#)
6. [Next steps](#)

Prerequisites

To set up and run this example, you must first complete these tasks:

- Set up the project environment to run these Node TypeScript examples, and install the required AWS SDK for JavaScript and third-party modules. Follow the instructions on [GitHub](#).
- Create a shared configurations file with your user credentials. For more information about providing a shared credentials file, see [Shared config and credentials files](#) in the *AWS SDKs and Tools Reference Guide*.

Important

This example uses ECMAScript6 (ES6). This requires Node.js version 13.x or higher. To download and install the latest version of Node.js, see [Node.js downloads](#).

However, if you prefer to use CommonJS syntax, please refer to [JavaScript ES6/CommonJS syntax](#).

Create the AWS resources

This tutorial requires the following resources.

- An unauthenticated IAM role with permissions for Amazon SQS.
- A FIFO Amazon SQS Queue named **Message.fifo** - for information about creating a queue, see [Creating an Amazon SQS queue](#).

You can create these resources manually, but we recommend provisioning these resources using the AWS CloudFormation (AWS CloudFormation) as described in this tutorial.

Note

The AWS CloudFormation is a software development framework that enables you to define cloud application resources. For more information, see the [AWS CloudFormation User Guide](#).

Create the AWS resources using the AWS CloudFormation

AWS CloudFormation enables you to create and provision AWS infrastructure deployments predictably and repeatedly. For more information about AWS CloudFormation, see the [AWS CloudFormation User Guide](#).

To create the AWS CloudFormation stack using the AWS CLI:

1. Install and configure the AWS CLI following the instructions in the [AWS CLI User Guide](#).
2. Create a file named `setup.yaml` in the root directory of your project folder, and copy the content [here on GitHub](#) into it.

Note

The AWS CloudFormation template was generated using the AWS CDK available [here on GitHub](#). For more information about the AWS CDK, see the [AWS Cloud Development Kit \(AWS CDK\) Developer Guide](#).

3. Run the following command from the command line, replacing `STACK_NAME` with a unique name for the stack.

Important

The stack name must be unique within an AWS Region and AWS account. You can specify up to 128 characters, and numbers and hyphens are allowed.

```
aws cloudformation create-stack --stack-name STACK_NAME --template-body file://  
setup.yaml --capabilities CAPABILITY_IAM
```

For more information on the `create-stack` command parameters, see the [AWS CLI Command Reference guide](#), and the [AWS CloudFormation User Guide](#).

To view the resources created, open AWS CloudFormation in the AWS management console, choose the stack, and select the **Resources** tab.

Understand the AWS Messaging application

To send a message to a SQS queue, enter the message into the application and choose Send.

After the message is sent, the application displays the message.

You can choose **Purge** to purge the messages from the Amazon SQS queue. This results in an empty queue, and no messages are displayed in the application.

The following describes how the application handles a message:

- The user selects their name and enters their message, and submits the message, which initiates the `pushMessage` function.
- `pushMessage` retrieves the Amazon SQS Queue Url, and then sends a message with a unique message ID value (a GUID) the message text, and the user to the Amazon SQS Queue.
- `pushMessage` retrieves the messages from the Amazon SQS Queue, extracts the user and message for each message, and displays the messages.
- The user can purge the messages, which delete the messages from the Amazon SQS Queue and from the user interface.

Create the HTML page

Now you create the HTML files that are required for the application's graphical user interface (GUI). Create a file named `index.html`. Copy and paste the code below in to `index.html`. This HTML references `main.js`. This is a bundled version of `index.js`, which includes the required AWS SDK for JavaScript modules.

```
<!doctype html>
<html
  xmlns:th="http://www.thymeleaf.org"
  xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-springsecurity3"
>
  <head>
    <meta charset="utf-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <link rel="icon" href="./images/favicon.ico" />
    <link
      rel="stylesheet"
      href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
    />
    <link rel="stylesheet" href="./css/styles.css" />
    <script src="https://code.jquery.com/jquery-1.12.4.min.js"></script>
    <script src="https://code.jquery.com/ui/1.11.4/jquery-ui.min.js"></script>
    <script src="./js/main.js"></script>
    <style>
      .messageelement {
        margin: auto;
        border: 2px solid #dedede;
      }
    </style>
  </head>
</html>
```



```
    background-color: #d7d1d0;
    border-radius: 5px;
    max-width: 800px;
    padding: 10px;
    margin: 10px 0;
  }

  .messageelement::after {
    content: "";
    clear: both;
    display: table;
  }

  .messageelement img {
    float: left;
    max-width: 60px;
    width: 100%;
    margin-right: 20px;
    border-radius: 50%;
  }

  .messageelement img.right {
    float: right;
    margin-left: 20px;
    margin-right: 0;
  }
</style>
</head>
<body>
  <div class="container">
    <h2>AWS Sample Messaging Application</h2>
    <div id="messages"></div>

    <div class="input-group mb-3">
      <div class="input-group-prepend">
        <span class="input-group-text" id="basic-addon1">Sender:</span>
      </div>
      <select name="cars" id="username">
        <option value="Scott">Brian</option>
        <option value="Tricia">Tricia</option>
      </select>
    </div>

    <div class="input-group">
```

```

<div class="input-group-prepend">
  <span class="input-group-text">Message:</span>
</div>
<textarea
  class="form-control"
  id="textarea"
  aria-label="With textarea"
></textarea>
<button
  type="button"
  onclick="pushMessage()"
  id="send"
  class="btn btn-success"
>
  Send
</button>
<button
  type="button"
  onclick="purge()"
  id="refresh"
  class="btn btn-success"
>
  Purge
</button>
</div>
<!-- All of these child items are hidden and only displayed in a FancyBox
----->
<div id="hide" style="display: none">
  <div id="base" class="messageelement">
    
    <p id="text">Excellent! So, what do you want to do today?</p>
    <span class="time-right">11:02</span>
  </div>
</div>
</div>
</body>
</html>

```

This code is also available [here on GitHub](#).

Creating the browser script

In this topic, you create a the browser script for the app. When you have created the browser script, you bundle it into a file called `main.js` as described in [Bundling the JavaScript](#).

Create a file named `index.js`. Copy and paste the code from [here on GitHub](#) into it.

This code is explained in the following sections:

1. [Configuration](#)
2. [populateChat](#)
3. [pushmessages](#)
4. [purge](#)

Configuration

First, create a `libs` directory create the required Amazon SQS client object by creating a files named `sqsClient.js`. Replace *REGION* and *IDENTITY_POOL_ID* in each.

Note

Use the ID of the Amazon Cognito identity pool you created in [Create the AWS resources](#).

```
import { CognitoIdentityClient } from "@aws-sdk/client-cognito-identity";
import { fromCognitoIdentityPool } from "@aws-sdk/credential-providers";
import { SQSClient } from "@aws-sdk/client-sqs";
const REGION = "REGION"; //e.g. "us-east-1"
const IdentityPoolId = "IDENTITY_POOL_ID";
const sqsClient = new SQSClient({
  region: REGION,
  credentials: fromCognitoIdentityPool({
    client: new CognitoIdentityClient({ region: REGION }),
    identityPoolId: IdentityPoolId
  }),
});
```

In the `index.js`, import the required AWS SDK for JavaScript modules and commands. Replace `SQS_QUEUE_NAME` with the name of the Amazon SQS Queue you created in the [Create the AWS resources](#).

```
import {
  GetQueueUrlCommand,
  SendMessageCommand,
  ReceiveMessageCommand,
  PurgeQueueCommand,
} from "@aws-sdk/client-sqs";
import { sqsClient } from "../libs/sqsClient.js";

const QueueName = "SQS_QUEUE_NAME"; // The Amazon SQS queue name, which must end
in .fifo for this example.
```

populateChat

The `populateChat` function onload automatically retrieves the URL for the Amazon SQS Queue, and retrieves all messages in the queue, and displays them.

```
$(function () {
  populateChat();
});

const populateChat = async () => {
  try {
    // Set the Amazon SQS Queue parameters.
    const queueParams = {
      QueueName: QueueName,
      Attributes: {
        DelaySeconds: "60",
        MessageRetentionPeriod: "86400",
      },
    };
  };
  // Get the Amazon SQS Queue URL.
  const data = await sqsClient.send(new GetQueueUrlCommand(queueParams));
  console.log("Success. The URL of the SQS Queue is: ", data.QueueUrl);
  // Set the parameters for retrieving the messages in the Amazon SQS Queue.
  var getMessageParams = {
    QueueUrl: data.QueueUrl,
    MaxNumberOfMessages: 10,
```

```
    MessageAttributeNames: ["All"],
    VisibilityTimeout: 20,
    WaitTimeSeconds: 20,
  };
  try {
    // Retrieve the messages from the Amazon SQS Queue.
    const data = await sqsClient.send(
      new ReceiveMessageCommand(getMessageParams)
    );
    console.log("Successfully retrieved messages", data.Messages);

    // Loop through messages for user and message body.
    var i;
    for (i = 0; i < data.Messages.length; i++) {
      const name = data.Messages[i].MessageAttributes.Name.StringValue;
      const body = data.Messages[i].Body;
      // Create the HTML for the message.
      var userText = body + "<br><br><b>" + name;
      var myTextNode = $("#base").clone();
      myTextNode.text(userText);
      var image_url;
      var n = name.localeCompare("Scott");
      if (n == 0) image_url = "./images/av1.png";
      else image_url = "./images/av2.png";
      var images_div =
        '';
      myTextNode.html(userText);
      myTextNode.append(images_div);

      // Add the message to the GUI.
      $("#messages").append(myTextNode);
    }
  } catch (err) {
    console.log("Error loading messages: ", err);
  }
} catch (err) {
  console.log("Error retrieving SQS queue URL: ", err);
}
};
```

Push messages

The user selects their name and enters their message, and submits the message, which initiates the `pushMessage` function. `pushMessage` retrieves the Amazon SQS Queue Url, and then sends a message with a unique message ID value (a GUID) the message text, and the user to the Amazon SQS Queue. It then retrieves all the messages from the Amazon SQS Queue and displays them.

```
const pushMessage = async () => {
  // Get and convert user and message input.
  var user = $("#username").val();
  var message = $("#textarea").val();

  // Create random deduplication ID.
  var dt = new Date().getTime();
  var uuid = "xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx".replace(/[xy]/g, function (
    c
  ) {
    var r = (dt + Math.random() * 16) % 16 | 0;
    dt = Math.floor(dt / 16);
    return (c == "x" ? r : (r & 0x3) | 0x8).toString(16);
  });

  try {
    // Set the Amazon SQS Queue parameters.
    const queueParams = {
      QueueName: QueueName,
      Attributes: {
        DelaySeconds: "60",
        MessageRetentionPeriod: "86400",
      },
    };
    const data = await sqsClient.send(new GetQueueUrlCommand(queueParams));
    console.log("Success. The URL of the SQS Queue is: ", data.QueueUrl);
    // Set the parameters for the message.
    var messageParams = {
      MessageAttributes: {
        Name: {
          DataType: "String",
          StringValue: user,
        },
      },
      MessageBody: message,
      MessageDeduplicationId: uuid,
    };
  }
}
```

```
    MessageGroupId: "GroupA",
    QueueUrl: data.QueueUrl,
  };
  const result = await sqsClient.send(new SendMessageCommand(messageParams));
  console.log("Success", result.MessageId);

  // Set the parameters for retrieving all messages in the SQS queue.
  var getMessageParams = {
    QueueUrl: data.QueueUrl,
    MaxNumberOfMessages: 10,
    MessageAttributeNames: ["All"],
    VisibilityTimeout: 20,
    WaitTimeSeconds: 20,
  };

  // Retrieve messages from SQS Queue.
  const final = await sqsClient.send(
    new ReceiveMessageCommand(getMessageParams)
  );
  console.log("Successfully retrieved", final.Messages);
  $("#messages").empty();
  // Loop through messages for user and message body.
  var i;
  for (i = 0; i < final.Messages.length; i++) {
    const name = final.Messages[i].MessageAttributes.Name.StringValue;
    const body = final.Messages[i].Body;
    // Create the HTML for the message.
    var userText = body + "<br><br><b>" + name;
    var myTextNode = $("#base").clone();
    myTextNode.text(userText);
    var image_url;
    var n = name.localeCompare("Scott");
    if (n == 0) image_url = "./images/av1.png";
    else image_url = "./images/av2.png";
    var images_div =
      '';
    myTextNode.html(userText);
    myTextNode.append(images_div);
    // Add the HTML to the GUI.
    $("#messages").append(myTextNode);
  }
} catch (err) {
```

```
    console.log("Error", err);
  }
};
// Make the function available to the browser window.
window.pushMessage = pushMessage;
```

Purge messages

`purge` deletes the messages from the Amazon SQS Queue and from the user interface.

```
// Delete the message from the Amazon SQS queue.
const purge = async () => {
  try {
    // Set the Amazon SQS Queue parameters.
    const queueParams = {
      QueueName: QueueName,
      Attributes: {
        DelaySeconds: "60",
        MessageRetentionPeriod: "86400",
      },
    };
    // Get the Amazon SQS Queue URL.
    const data = await sqsClient.send(new GetQueueUrlCommand(queueParams));
    console.log("Success", data.QueueUrl);
    // Delete all the messages in the Amazon SQS Queue.
    const result = await sqsClient.send(
      new PurgeQueueCommand({ QueueUrl: data.QueueUrl })
    );
    // Delete all the messages from the GUI.
    $("#messages").empty();
    console.log("Success. All messages deleted.", data);
  } catch (err) {
    console.log("Error", err);
  }
};


// Make the function available to the browser window.
window.purge = purge;
```

Bundling the JavaScript

This complete browser script code is available [here on GitHub](#).

Now use webpack to bundle the `index.js` and AWS SDK for JavaScript modules into a single file, `main.js`.

1. If you haven't already, follow the [Prerequisites](#) for this example to install webpack.

 **Note**

For information about *webpack*, see [Bundle applications with webpack](#).

2. Run the the following in the command line to bundle the JavaScript for this example into a file called `<index.js>` :

```
webpack index.js --mode development --target web --devtool false -o main.js
```

Next steps

Congratulations! You have created and deployed the AWS Messaging application that uses Amazon SQS. As stated at the beginning of this tutorial, be sure to terminate all of the resources you create while going through this tutorial to ensure that you're no longer charged for them. You can do this by deleting the AWS CloudFormation stack you created in the [Create the AWS resources](#) topic of this tutorial, as follows:

1. Open the [AWS CloudFormation in the AWS management console](#).
2. Open the **Stacks** page, and select the stack.
3. Choose **Delete**.

Use AWS Cloud9 with the AWS SDK for JavaScript

You can use AWS Cloud9 with the AWS SDK for JavaScript to write and run your JavaScript in the browser code—as well as write, run, and debug your Node.js code—using just a browser. AWS Cloud9 includes tools such as a code editor and terminal, plus a debugger for Node.js code.

Because the AWS Cloud9 IDE is cloud based, you can work on your projects from your office, home, or anywhere using an internet-connected machine. For general information about AWS Cloud9, see the [AWS Cloud9 User Guide](#).

The following steps describe how to set up AWS Cloud9 with the SDK for JavaScript.

Contents

- [Step 1: Set up your AWS account to use AWS Cloud9](#)
- [Step 2: Set up your AWS Cloud9 development environment](#)
- [Step 3: Set up the SDK for JavaScript](#)
 - [To set up the SDK for JavaScript for Node.js](#)
 - [To set up the SDK for JavaScript in the browser](#)
- [Step 4: Download example code](#)
- [Step 5: Run and debug example code](#)

Step 1: Set up your AWS account to use AWS Cloud9

Start to use AWS Cloud9 by signing in to the AWS Cloud9 console as an AWS Identity and Access Management (IAM) entity (for example, an IAM user) who has access permissions for AWS Cloud9 in your AWS account.

To set up an IAM entity in your AWS account to access AWS Cloud9, and to sign in to the AWS Cloud9 console, see [Team setup for AWS Cloud9](#) in the *AWS Cloud9 User Guide*.

Step 2: Set up your AWS Cloud9 development environment

After you sign in to the AWS Cloud9 console, use the console to create an AWS Cloud9 development environment. After you create the environment, AWS Cloud9 opens the IDE for that environment.

See [Creating an environment in AWS Cloud9](#) in the *AWS Cloud9 User Guide* for details.

Note

As you create your environment in the console for the first time, we recommend that you choose the option to **Create a new instance for environment (EC2)**. This option tells AWS Cloud9 to create an environment, launch an Amazon EC2 instance, and then connect the new instance to the new environment. This is the fastest way to begin using AWS Cloud9.

Step 3: Set up the SDK for JavaScript

After AWS Cloud9 opens the IDE for your development environment, follow one or both of the following procedures to use the IDE to set up the SDK for JavaScript in your environment.

To set up the SDK for JavaScript for Node.js

1. If the terminal isn't already open in the IDE, open it. To do this, on the menu bar in the IDE, choose **Window, New Terminal**.
2. Run the following command to use npm to install the Cloud9 client of the SDK for JavaScript.

```
npm install @aws-sdk/client-cloud9
```

If the IDE can't find npm, run the following commands, one at a time in the following order, to install npm. (These commands assume you chose the option to **Create a new instance for environment (EC2)**, earlier in this topic.)

Warning

AWS does not control the following code. Before you run it, be sure to verify its authenticity and integrity. More information about this code can be found in the [npm](#) (Node Version Manager) GitHub repository.

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.34.0/install.sh | bash #  
Download and install Node Version Manager (nvm).  
. ~/.bashrc #  
Activate nvm.
```

```
nvm install node #  
Use nvm to install npm (and Node.js at the same time).
```

To set up the SDK for JavaScript in the browser

To use the SDK for JavaScript in your HTML pages, use WebPack to bundle the required client modules and all required JavaScript functions into a single JavaScript file, and add it in a script tag in the <head> of your HTML pages. For example:

```
<script src=./main.js></script>
```

Note

For more information about Webpack, see [Bundle applications with webpack](#)

Step 4: Download example code

Use the terminal you opened in the previous step to download example code for the SDK for JavaScript into the AWS Cloud9 development environment. (If the terminal isn't already open in the IDE, open it by choosing **Window, New Terminal** on the menu bar in the IDE.)

To download the example code, run the following command. This command downloads a copy of all of the code examples used in the official AWS SDK documentation into your environment's root directory.

```
git clone https://github.com/awsdocs/aws-doc-sdk-examples.git
```

To find code examples for the SDK for JavaScript, use the **Environment** window to open the `ENVIRONMENT_NAME\aws-doc-sdk-examples\javascriptv3\example_code/src`, where `ENVIRONMENT_NAME` is the name of your AWS Cloud9 development environment.

To learn how to work with these and other code examples, see [SDK for JavaScript code examples](#).

Step 5: Run and debug example code

To run code in your AWS Cloud9 development environment, see [Run your code](#) in the *AWS Cloud9 User Guide*.

To debug Node.js code, see [Debug your code](#) in the *AWS Cloud9 User Guide*.

SDK for JavaScript (v3) code examples

The code examples in this topic show you how to use the AWS SDK for JavaScript (v3) with AWS.

Basics are code examples that show you how to perform the essential operations within a service.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Scenarios are code examples that show you how to accomplish specific tasks by calling multiple functions within a service or combined with other AWS services.

Services

- [API Gateway examples using SDK for JavaScript \(v3\)](#)
- [Aurora examples using SDK for JavaScript \(v3\)](#)
- [Auto Scaling examples using SDK for JavaScript \(v3\)](#)
- [Amazon Bedrock examples using SDK for JavaScript \(v3\)](#)
- [Amazon Bedrock Runtime examples using SDK for JavaScript \(v3\)](#)
- [Agents for Amazon Bedrock examples using SDK for JavaScript \(v3\)](#)
- [Agents for Amazon Bedrock Runtime examples using SDK for JavaScript \(v3\)](#)
- [CloudWatch examples using SDK for JavaScript \(v3\)](#)
- [CloudWatch Events examples using SDK for JavaScript \(v3\)](#)
- [CloudWatch Logs examples using SDK for JavaScript \(v3\)](#)
- [CodeBuild examples using SDK for JavaScript \(v3\)](#)
- [Amazon Cognito Identity Provider examples using SDK for JavaScript \(v3\)](#)
- [Amazon Comprehend examples using SDK for JavaScript \(v3\)](#)
- [Amazon DocumentDB examples using SDK for JavaScript \(v3\)](#)
- [DynamoDB examples using SDK for JavaScript \(v3\)](#)
- [Amazon EC2 examples using SDK for JavaScript \(v3\)](#)
- [Elastic Load Balancing - Version 2 examples using SDK for JavaScript \(v3\)](#)
- [EventBridge examples using SDK for JavaScript \(v3\)](#)

- [AWS Glue examples using SDK for JavaScript \(v3\)](#)
- [HealthImaging examples using SDK for JavaScript \(v3\)](#)
- [IAM examples using SDK for JavaScript \(v3\)](#)
- [Kinesis examples using SDK for JavaScript \(v3\)](#)
- [Lambda examples using SDK for JavaScript \(v3\)](#)
- [Amazon Lex examples using SDK for JavaScript \(v3\)](#)
- [Amazon MSK examples using SDK for JavaScript \(v3\)](#)
- [Amazon Personalize examples using SDK for JavaScript \(v3\)](#)
- [Amazon Personalize Events examples using SDK for JavaScript \(v3\)](#)
- [Amazon Personalize Runtime examples using SDK for JavaScript \(v3\)](#)
- [Amazon Pinpoint examples using SDK for JavaScript \(v3\)](#)
- [Amazon Polly examples using SDK for JavaScript \(v3\)](#)
- [Amazon RDS examples using SDK for JavaScript \(v3\)](#)
- [Amazon RDS Data Service examples using SDK for JavaScript \(v3\)](#)
- [Amazon Redshift examples using SDK for JavaScript \(v3\)](#)
- [Amazon Rekognition examples using SDK for JavaScript \(v3\)](#)
- [Amazon S3 examples using SDK for JavaScript \(v3\)](#)
- [S3 Glacier examples using SDK for JavaScript \(v3\)](#)
- [SageMaker examples using SDK for JavaScript \(v3\)](#)
- [Secrets Manager examples using SDK for JavaScript \(v3\)](#)
- [Amazon SES examples using SDK for JavaScript \(v3\)](#)
- [Amazon SNS examples using SDK for JavaScript \(v3\)](#)
- [Amazon SQS examples using SDK for JavaScript \(v3\)](#)
- [Step Functions examples using SDK for JavaScript \(v3\)](#)
- [AWS STS examples using SDK for JavaScript \(v3\)](#)
- [AWS Support examples using SDK for JavaScript \(v3\)](#)
- [Amazon Textract examples using SDK for JavaScript \(v3\)](#)
- [Amazon Transcribe examples using SDK for JavaScript \(v3\)](#)
- [Amazon Translate examples using SDK for JavaScript \(v3\)](#)

API Gateway examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with API Gateway.

Scenarios are code examples that show you how to accomplish specific tasks by calling multiple functions within a service or combined with other AWS services.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Topics

- [Scenarios](#)

Scenarios

Create a serverless application to manage photos

The following code example shows how to create a serverless application that lets users manage photos using labels.

SDK for JavaScript (v3)

Shows how to develop a photo asset management application that detects labels in images using Amazon Rekognition and stores them for later retrieval.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

For a deep dive into the origin of this example see the post on [AWS Community](#).

Services used in this example

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Use API Gateway to invoke a Lambda function

The following code example shows how to create an AWS Lambda function invoked by Amazon API Gateway.

SDK for JavaScript (v3)

Shows how to create an AWS Lambda function by using the Lambda JavaScript runtime API. This example invokes different AWS services to perform a specific use case. This example demonstrates how to create a Lambda function invoked by Amazon API Gateway that scans an Amazon DynamoDB table for work anniversaries and uses Amazon Simple Notification Service (Amazon SNS) to send a text message to your employees that congratulates them at their one year anniversary date.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

This example is also available in the [AWS SDK for JavaScript v3 developer guide](#).

Services used in this example

- API Gateway
- DynamoDB
- Lambda
- Amazon SNS

Aurora examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with Aurora.

Scenarios are code examples that show you how to accomplish specific tasks by calling multiple functions within a service or combined with other AWS services.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Topics

- [Scenarios](#)

Scenarios

Create an Aurora Serverless work item tracker

The following code example shows how to create a web application that tracks work items in an Amazon Aurora Serverless database and uses Amazon Simple Email Service (Amazon SES) to send reports.

SDK for JavaScript (v3)

Shows how to use the AWS SDK for JavaScript (v3) to create a web application that tracks work items in an Amazon Aurora database and emails reports by using Amazon Simple Email Service (Amazon SES). This example uses a front end built with React.js to interact with an Express Node.js backend.

- Integrate a React.js web application with AWS services.
- List, add, and update items in an Aurora table.
- Send an email report of filtered work items by using Amazon SES.
- Deploy and manage example resources with the included AWS CloudFormation script.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- Aurora
- Amazon RDS
- Amazon RDS Data Service
- Amazon SES

Auto Scaling examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with Auto Scaling.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Scenarios are code examples that show you how to accomplish specific tasks by calling multiple functions within a service or combined with other AWS services.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Topics

- [Actions](#)
- [Scenarios](#)

Actions

AttachLoadBalancerTargetGroups

The following code example shows how to use `AttachLoadBalancerTargetGroups`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const client = new AutoScalingClient({});
await client.send(
  new AttachLoadBalancerTargetGroupsCommand({
    AutoScalingGroupName: NAMES.autoScalingGroupName,
    TargetGroupARNs: [state.targetGroupArn],
  }),
);
```

- For API details, see [AttachLoadBalancerTargetGroups](#) in *AWS SDK for JavaScript API Reference*.

Scenarios

Build and manage a resilient service

The following code example shows how to create a load-balanced web service that returns book, movie, and song recommendations. The example shows how the service responds to failures, and how to restructure the service for more resilience when failures occur.

- Use an Amazon EC2 Auto Scaling group to create Amazon Elastic Compute Cloud (Amazon EC2) instances based on a launch template and to keep the number of instances in a specified range.
- Handle and distribute HTTP requests with Elastic Load Balancing.
- Monitor the health of instances in an Auto Scaling group and forward requests only to healthy instances.
- Run a Python web server on each EC2 instance to handle HTTP requests. The web server responds with recommendations and health checks.
- Simulate a recommendation service with an Amazon DynamoDB table.
- Control web server response to requests and health checks by updating AWS Systems Manager parameters.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Run the interactive scenario at a command prompt.

```
#!/usr/bin/env node
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

import {
  Scenario,
  parseScenarioArgs,
} from "@aws-doc-sdk-examples/lib/scenario/index.js";

/**
 * The workflow steps are split into three stages:
 * - deploy
 * - demo
 * - destroy
 *
 * Each of these stages has a corresponding file prefixed with steps-*.
 */
```

```
import { deploySteps } from "./steps-deploy.js";
import { demoSteps } from "./steps-demo.js";
import { destroySteps } from "./steps-destroy.js";

/**
 * The context is passed to every scenario. Scenario steps
 * will modify the context.
 */
const context = {};

/**
 * Three Scenarios are created for the workflow. A Scenario is an orchestration
 class
 * that simplifies running a series of steps.
 */
export const scenarios = {
  // Deploys all resources necessary for the workflow.
  deploy: new Scenario("Resilient Workflow - Deploy", deploySteps, context),
  // Demonstrates how a fragile web service can be made more resilient.
  demo: new Scenario("Resilient Workflow - Demo", demoSteps, context),
  // Destroys the resources created for the workflow.
  destroy: new Scenario("Resilient Workflow - Destroy", destroySteps, context),
};

// Call function if run directly
import { fileURLToPath } from "url";

if (process.argv[1] === fileURLToPath(import.meta.url)) {
  parseScenarioArgs(scenarios);
}
```

Create steps to deploy all of the resources.

```
import { join } from "node:path";
import { readFileSync, writeFileSync } from "node:fs";
import axios from "axios";

import {
  BatchWriteItemCommand,
  CreateTableCommand,
  DynamoDBClient,
  waitUntilTableExists,
```

```
} from "@aws-sdk/client-dynamodb";
import {
  EC2Client,
  CreateKeyPairCommand,
  CreateLaunchTemplateCommand,
  DescribeAvailabilityZonesCommand,
  DescribeVpcsCommand,
  DescribeSubnetsCommand,
  DescribeSecurityGroupsCommand,
  AuthorizeSecurityGroupIngressCommand,
} from "@aws-sdk/client-ec2";
import {
  IAMClient,
  CreatePolicyCommand,
  CreateRoleCommand,
  CreateInstanceProfileCommand,
  AddRoleToInstanceProfileCommand,
  AttachRolePolicyCommand,
  waitUntilInstanceProfileExists,
} from "@aws-sdk/client-iam";
import { SSMClient, GetParameterCommand } from "@aws-sdk/client-ssm";
import {
  CreateAutoScalingGroupCommand,
  AutoScalingClient,
  AttachLoadBalancerTargetGroupsCommand,
} from "@aws-sdk/client-auto-scaling";
import {
  CreateListenerCommand,
  CreateLoadBalancerCommand,
  CreateTargetGroupCommand,
  ElasticLoadBalancingV2Client,
  waitUntilLoadBalancerAvailable,
} from "@aws-sdk/client-elastic-load-balancing-v2";

import {
  ScenarioOutput,
  ScenarioInput,
  ScenarioAction,
} from "@aws-doc-sdk-examples/lib/scenario/index.js";
import { saveState } from "@aws-doc-sdk-examples/lib/scenario/steps-common.js";
import { retry } from "@aws-doc-sdk-examples/lib/utils/util-timers.js";

import { MESSAGES, NAMES, RESOURCES_PATH, ROOT } from "./constants.js";
import { initParamsSteps } from "./steps-reset-params.js";
```

```
/**
 * @type {import('@aws-doc-sdk-examples/lib/scenario.js').Step[]}
 */
export const deploySteps = [
  new ScenarioOutput("introduction", MESSAGES.introduction, { header: true }),
  new ScenarioInput("confirmDeployment", MESSAGES.confirmDeployment, {
    type: "confirm",
  }),
  new ScenarioAction(
    "handleConfirmDeployment",
    (c) => c.confirmDeployment === false && process.exit(),
  ),
  new ScenarioOutput(
    "creatingTable",
    MESSAGES.creatingTable.replace("${TABLE_NAME}", NAMES.tableName),
  ),
  new ScenarioAction("createTable", async () => {
    const client = new DynamoDBClient({});
    await client.send(
      new CreateTableCommand({
        TableName: NAMES.tableName,
        ProvisionedThroughput: {
          ReadCapacityUnits: 5,
          WriteCapacityUnits: 5,
        },
        AttributeDefinitions: [
          {
            AttributeName: "MediaType",
            AttributeType: "S",
          },
          {
            AttributeName: "ItemId",
            AttributeType: "N",
          },
        ],
        KeySchema: [
          {
            AttributeName: "MediaType",
            KeyType: "HASH",
          },
          {
            AttributeName: "ItemId",
            KeyType: "RANGE",
          },
        ],
      })
    );
  })
];
```

```

    },
  ],
}),
);
await waitUntilTableExists({ client }, { TableName: NAMES.tableName });
}),
new ScenarioOutput(
  "createdTable",
  MESSAGES.createdTable.replace("${TABLE_NAME}", NAMES.tableName),
),
new ScenarioOutput(
  "populatingTable",
  MESSAGES.populatingTable.replace("${TABLE_NAME}", NAMES.tableName),
),
new ScenarioAction("populateTable", () => {
  const client = new DynamoDBClient({});
  /**
   * @type {{ default: import("@aws-sdk/client-dynamodb").PutRequest['Item'][] }}
   */
  const recommendations = JSON.parse(
    readFileSync(join(RESOURCES_PATH, "recommendations.json")),
  );

  return client.send(
    new BatchWriteItemCommand({
      RequestItems: {
        [NAMES.tableName]: recommendations.map((item) => ({
          PutRequest: { Item: item },
        })),
      },
    }),
  );
}),
new ScenarioOutput(
  "populatedTable",
  MESSAGES.populatedTable.replace("${TABLE_NAME}", NAMES.tableName),
),
new ScenarioOutput(
  "creatingKeyPair",
  MESSAGES.creatingKeyPair.replace("${KEY_PAIR_NAME}", NAMES.keyPairName),
),
new ScenarioAction("createKeyPair", async () => {
  const client = new EC2Client({});
  const { KeyMaterial } = await client.send(

```



```

    new CreateKeyPairCommand({
      KeyName: NAMES.keyPairName,
    }),
  );

  writeFileSync(`${NAMES.keyPairName}.pem`, KeyMaterial, { mode: 0o600 });
}),
new ScenarioOutput(
  "createdKeyPair",
  MESSAGES.createdKeyPair.replace("${KEY_PAIR_NAME}", NAMES.keyPairName),
),
new ScenarioOutput(
  "creatingInstancePolicy",
  MESSAGES.creatingInstancePolicy.replace(
    "${INSTANCE_POLICY_NAME}",
    NAMES.instancePolicyName,
  ),
),
new ScenarioAction("createInstancePolicy", async (state) => {
  const client = new IAMClient({});
  const {
    Policy: { Arn },
  } = await client.send(
    new CreatePolicyCommand({
      PolicyName: NAMES.instancePolicyName,
      PolicyDocument: readFileSync(
        join(RESOURCES_PATH, "instance_policy.json"),
      ),
    }),
  );
  state.instancePolicyArn = Arn;
}),
new ScenarioOutput("createdInstancePolicy", (state) =>
  MESSAGES.createdInstancePolicy
    .replace("${INSTANCE_POLICY_NAME}", NAMES.instancePolicyName)
    .replace("${INSTANCE_POLICY_ARN}", state.instancePolicyArn),
),
new ScenarioOutput(
  "creatingInstanceRole",
  MESSAGES.creatingInstanceRole.replace(
    "${INSTANCE_ROLE_NAME}",
    NAMES.instanceRoleName,
  ),
),
),

```

```
new ScenarioAction("createInstanceRole", () => {
  const client = new IAMClient({});
  return client.send(
    new CreateRoleCommand({
      RoleName: NAMES.instanceRoleName,
      AssumeRolePolicyDocument: readFileSync(
        join(ROOT, "assume-role-policy.json"),
      ),
    }),
  );
}),
new ScenarioOutput(
  "createdInstanceRole",
  MESSAGES.createdInstanceRole.replace(
    "${INSTANCE_ROLE_NAME}",
    NAMES.instanceRoleName,
  ),
),
new ScenarioOutput(
  "attachingPolicyToRole",
  MESSAGES.attachingPolicyToRole
    .replace("${INSTANCE_ROLE_NAME}", NAMES.instanceRoleName)
    .replace("${INSTANCE_POLICY_NAME}", NAMES.instancePolicyName),
),
new ScenarioAction("attachPolicyToRole", async (state) => {
  const client = new IAMClient({});
  await client.send(
    new AttachRolePolicyCommand({
      RoleName: NAMES.instanceRoleName,
      PolicyArn: state.instancePolicyArn,
    }),
  );
}),
new ScenarioOutput(
  "attachedPolicyToRole",
  MESSAGES.attachedPolicyToRole
    .replace("${INSTANCE_POLICY_NAME}", NAMES.instancePolicyName)
    .replace("${INSTANCE_ROLE_NAME}", NAMES.instanceRoleName),
),
new ScenarioOutput(
  "creatingInstanceProfile",
  MESSAGES.creatingInstanceProfile.replace(
    "${INSTANCE_PROFILE_NAME}",
    NAMES.instanceProfileName,
```

```

    ),
  ),
  new ScenarioAction("createInstanceProfile", async (state) => {
    const client = new IAMClient({});
    const {
      InstanceProfile: { Arn },
    } = await client.send(
      new CreateInstanceProfileCommand({
        InstanceProfileName: NAMES.instanceProfileName,
      }),
    );
    state.instanceProfileArn = Arn;

    await waitUntilInstanceProfileExists(
      { client },
      { InstanceProfileName: NAMES.instanceProfileName },
    );
  }),
  new ScenarioOutput("createdInstanceProfile", (state) =>
    MESSAGES.createdInstanceProfile
      .replace("${INSTANCE_PROFILE_NAME}", NAMES.instanceProfileName)
      .replace("${INSTANCE_PROFILE_ARN}", state.instanceProfileArn),
  ),
  new ScenarioOutput(
    "addingRoleToInstanceProfile",
    MESSAGES.addingRoleToInstanceProfile
      .replace("${INSTANCE_PROFILE_NAME}", NAMES.instanceProfileName)
      .replace("${INSTANCE_ROLE_NAME}", NAMES.instanceRoleName),
  ),
  new ScenarioAction("addRoleToInstanceProfile", () => {
    const client = new IAMClient({});
    return client.send(
      new AddRoleToInstanceProfileCommand({
        RoleName: NAMES.instanceRoleName,
        InstanceProfileName: NAMES.instanceProfileName,
      }),
    );
  }),
  new ScenarioOutput(
    "addedRoleToInstanceProfile",
    MESSAGES.addedRoleToInstanceProfile
      .replace("${INSTANCE_PROFILE_NAME}", NAMES.instanceProfileName)
      .replace("${INSTANCE_ROLE_NAME}", NAMES.instanceRoleName),
  ),
),

```

```
...initParamsSteps,
new ScenarioOutput("creatingLaunchTemplate", MESSAGES.creatingLaunchTemplate),
new ScenarioAction("createLaunchTemplate", async () => {
  const ssmClient = new SSMClient({});
  const { Parameter } = await ssmClient.send(
    new GetParameterCommand({
      Name: "/aws/service/ami-amazon-linux-latest/amzn2-ami-hvm-x86_64-gp2",
    }),
  );
  const ec2Client = new EC2Client({});
  await ec2Client.send(
    new CreateLaunchTemplateCommand({
      LaunchTemplateName: NAMES.launchTemplateName,
      LaunchTemplateData: {
        InstanceType: "t3.micro",
        ImageId: Parameter.Value,
        IamInstanceProfile: { Name: NAMES.instanceProfileName },
        UserData: readFileSync(
          join(RESOURCES_PATH, "server_startup_script.sh"),
        ).toString("base64"),
        KeyName: NAMES.keyPairName,
      },
    }),
  );
},
new ScenarioOutput(
  "createdLaunchTemplate",
  MESSAGES.createdLaunchTemplate.replace(
    "${LAUNCH_TEMPLATE_NAME}",
    NAMES.launchTemplateName,
  ),
),
new ScenarioOutput(
  "creatingAutoScalingGroup",
  MESSAGES.creatingAutoScalingGroup.replace(
    "${AUTO_SCALING_GROUP_NAME}",
    NAMES.autoScalingGroupName,
  ),
),
new ScenarioAction("createAutoScalingGroup", async (state) => {
  const ec2Client = new EC2Client({});
  const { AvailabilityZones } = await ec2Client.send(
    new DescribeAvailabilityZonesCommand({}),
  );
```

```

state.availabilityZoneNames = AvailabilityZones.map((az) => az.ZoneName);
const autoScalingClient = new AutoScalingClient({});
await retry({ intervalInMs: 1000, maxRetries: 30 }, () =>
  autoScalingClient.send(
    new CreateAutoScalingGroupCommand({
      AvailabilityZones: state.availabilityZoneNames,
      AutoScalingGroupName: NAMES.autoScalingGroupName,
      LaunchTemplate: {
        LaunchTemplateName: NAMES.launchTemplateName,
        Version: "$Default",
      },
      MinSize: 3,
      MaxSize: 3,
    }),
  ),
);
}),
new ScenarioOutput(
  "createdAutoScalingGroup",
  /**
   * @param {{ availabilityZoneNames: string[] }} state
   */
  (state) =>
    MESSAGES.createdAutoScalingGroup
      .replace("${AUTO_SCALING_GROUP_NAME}", NAMES.autoScalingGroupName)
      .replace(
        "${AVAILABILITY_ZONE_NAMES}",
        state.availabilityZoneNames.join(", "),
      ),
  ),
new ScenarioInput("confirmContinue", MESSAGES.confirmContinue, {
  type: "confirm",
}),
new ScenarioOutput("loadBalancer", MESSAGES.loadBalancer),
new ScenarioOutput("gettingVpc", MESSAGES.gettingVpc),
new ScenarioAction("getVpc", async (state) => {
  const client = new EC2Client({});
  const { Vpcs } = await client.send(
    new DescribeVpcsCommand({
      Filters: [{ Name: "is-default", Values: ["true"] }],
    }),
  );
});
state.defaultVpc = Vpcs[0].VpcId;
}),

```

```

new ScenarioOutput("gotVpc", (state) =>
  MESSAGES.gotVpc.replace("${VPC_ID}", state.defaultVpc),
),
new ScenarioOutput("gettingSubnets", MESSAGES.gettingSubnets),
new ScenarioAction("getSubnets", async (state) => {
  const client = new EC2Client({});
  const { Subnets } = await client.send(
    new DescribeSubnetsCommand({
      Filters: [
        { Name: "vpc-id", Values: [state.defaultVpc] },
        { Name: "availability-zone", Values: state.availabilityZoneNames },
        { Name: "default-for-az", Values: ["true"] },
      ],
    }),
  );
  state.subnets = Subnets.map((subnet) => subnet.SubnetId);
}),
new ScenarioOutput(
  "gotSubnets",
  /**
   * @param {{ subnets: string[] }} state
   */
  (state) =>
    MESSAGES.gotSubnets.replace("${SUBNETS}", state.subnets.join(", ")),
),
new ScenarioOutput(
  "creatingLoadBalancerTargetGroup",
  MESSAGES.creatingLoadBalancerTargetGroup.replace(
    "${TARGET_GROUP_NAME}",
    NAMES.loadBalancerTargetGroupName,
  ),
),
new ScenarioAction("createLoadBalancerTargetGroup", async (state) => {
  const client = new ElasticLoadBalancingV2Client({});
  const { TargetGroups } = await client.send(
    new CreateTargetGroupCommand({
      Name: NAMES.loadBalancerTargetGroupName,
      Protocol: "HTTP",
      Port: 80,
      HealthCheckPath: "/healthcheck",
      HealthCheckIntervalSeconds: 10,
      HealthCheckTimeoutSeconds: 5,
      HealthyThresholdCount: 2,
      UnhealthyThresholdCount: 2,
    })
  );
});

```

```

        VpcId: state.defaultVpc,
    )),
    );
    const targetGroup = TargetGroups[0];
    state.targetGroupArn = targetGroup.TargetGroupArn;
    state.targetGroupProtocol = targetGroup.Protocol;
    state.targetGroupPort = targetGroup.Port;
  )),
  new ScenarioOutput(
    "createdLoadBalancerTargetGroup",
    MESSAGES.createdLoadBalancerTargetGroup.replace(
      "${TARGET_GROUP_NAME}",
      NAMES.loadBalancerTargetGroupName,
    ),
  ),
  new ScenarioOutput(
    "creatingLoadBalancer",
    MESSAGES.creatingLoadBalancer.replace("${LB_NAME}", NAMES.loadBalancerName),
  ),
  new ScenarioAction("createLoadBalancer", async (state) => {
    const client = new ElasticLoadBalancingV2Client({});
    const { LoadBalancers } = await client.send(
      new CreateLoadBalancerCommand({
        Name: NAMES.loadBalancerName,
        Subnets: state.subnets,
      }),
    );
    state.loadBalancerDns = LoadBalancers[0].DNSName;
    state.loadBalancerArn = LoadBalancers[0].LoadBalancerArn;
    await waitUntilLoadBalancerAvailable(
      { client },
      { Names: [NAMES.loadBalancerName] },
    );
  )),
  new ScenarioOutput("createdLoadBalancer", (state) =>
    MESSAGES.createdLoadBalancer
      .replace("${LB_NAME}", NAMES.loadBalancerName)
      .replace("${DNS_NAME}", state.loadBalancerDns),
  ),
  new ScenarioOutput(
    "creatingListener",
    MESSAGES.creatingLoadBalancerListener
      .replace("${LB_NAME}", NAMES.loadBalancerName)
      .replace("${TARGET_GROUP_NAME}", NAMES.loadBalancerTargetGroupName),
  ),

```

```
),
new ScenarioAction("createListener", async (state) => {
  const client = new ElasticLoadBalancingV2Client({});
  const { Listeners } = await client.send(
    new CreateListenerCommand({
      LoadBalancerArn: state.loadBalancerArn,
      Protocol: state.targetGroupProtocol,
      Port: state.targetGroupPort,
      DefaultActions: [
        { Type: "forward", TargetGroupArn: state.targetGroupArn },
      ],
    })
  );
  const listener = Listeners[0];
  state.loadBalancerListenerArn = listener.ListenerArn;
}),
new ScenarioOutput("createdListener", (state) =>
  MESSAGES.createdLoadBalancerListener.replace(
    "${LB_LISTENER_ARN}",
    state.loadBalancerListenerArn,
  ),
),
new ScenarioOutput(
  "attachingLoadBalancerTargetGroup",
  MESSAGES.attachingLoadBalancerTargetGroup
    .replace("${TARGET_GROUP_NAME}", NAMES.loadBalancerTargetGroupName)
    .replace("${AUTO_SCALING_GROUP_NAME}", NAMES.autoScalingGroupName),
),
new ScenarioAction("attachLoadBalancerTargetGroup", async (state) => {
  const client = new AutoScalingClient({});
  await client.send(
    new AttachLoadBalancerTargetGroupsCommand({
      AutoScalingGroupName: NAMES.autoScalingGroupName,
      TargetGroupARNs: [state.targetGroupArn],
    })
  );
}),
new ScenarioOutput(
  "attachedLoadBalancerTargetGroup",
  MESSAGES.attachedLoadBalancerTargetGroup,
),
new ScenarioOutput("verifyingInboundPort", MESSAGES.verifyingInboundPort),
new ScenarioAction(
  "verifyInboundPort",
```



```

/**
 *
 * @param {{ defaultSecurityGroup: import('@aws-sdk/client-ec2').SecurityGroup}}
state
 */
async (state) => {
  const client = new EC2Client({});
  const { SecurityGroups } = await client.send(
    new DescribeSecurityGroupsCommand({
      Filters: [{ Name: "group-name", Values: ["default"] }],
    }),
  );
  if (!SecurityGroups) {
    state.verifyInboundPortError = new Error(MESSAGES.noSecurityGroups);
  }
  state.defaultSecurityGroup = SecurityGroups[0];

  /**
   * @type {string}
   */
  const ipResponse = (await axios.get("http://checkip.amazonaws.com")).data;
  state.myIp = ipResponse.trim();
  const myIpRules = state.defaultSecurityGroup.IpPermissions.filter(
    ({ IpRanges }) =>
      IpRanges.some(
        ({ CidrIp }) =>
          CidrIp.startsWith(state.myIp) || CidrIp === "0.0.0.0/0",
      ),
  )
  .filter(({ IpProtocol }) => IpProtocol === "tcp")
  .filter(({ FromPort }) => FromPort === 80);

  state.myIpRules = myIpRules;
},
),
new ScenarioOutput(
  "verifiedInboundPort",
  /**
   * @param {{ myIpRules: any[] }} state
   */
  (state) => {
    if (state.myIpRules.length > 0) {
      return MESSAGES.foundIpRules.replace(
        "${IP_RULES}",

```

```

        JSON.stringify(state.myIpRules, null, 2),
    );
    } else {
        return MESSAGES.noIpRules;
    }
},
),
new ScenarioInput(
    "shouldAddInboundRule",
    /**
     * @param {{ myIpRules: any[] }} state
     */
    (state) => {
        if (state.myIpRules.length > 0) {
            return false;
        } else {
            return MESSAGES.noIpRules;
        }
    },
    { type: "confirm" },
),
new ScenarioAction(
    "addInboundRule",
    /**
     * @param {{ defaultSecurityGroup: import('@aws-sdk/client-ec2').SecurityGroup }} state
     */
    async (state) => {
        if (!state.shouldAddInboundRule) {
            return;
        }

        const client = new EC2Client({});
        await client.send(
            new AuthorizeSecurityGroupIngressCommand({
                GroupId: state.defaultSecurityGroup.GroupId,
                CidrIp: `${state.myIp}/32`,
                FromPort: 80,
                ToPort: 80,
                IpProtocol: "tcp",
            }),
        );
    },
),
),
),

```

```

new ScenarioOutput("addedInboundRule", (state) => {
  if (state.shouldAddInboundRule) {
    return MESSAGES.addedInboundRule.replace("${IP_ADDRESS}", state.myIp);
  } else {
    return false;
  }
}),
new ScenarioOutput("verifyingEndpoint", (state) =>
  MESSAGES.verifyingEndpoint.replace("${DNS_NAME}", state.loadBalancerDns),
),
new ScenarioAction("verifyEndpoint", async (state) => {
  try {
    const response = await retry({ intervalInMs: 2000, maxRetries: 30 }, () =>
      axios.get(`http://${state.loadBalancerDns}`),
    );
    state.endpointResponse = JSON.stringify(response.data, null, 2);
  } catch (e) {
    state.verifyEndpointError = e;
  }
}),
new ScenarioOutput("verifiedEndpoint", (state) => {
  if (state.verifyEndpointError) {
    console.error(state.verifyEndpointError);
  } else {
    return MESSAGES.verifiedEndpoint.replace(
      "${ENDPOINT_RESPONSE}",
      state.endpointResponse,
    );
  }
}),
saveState,
];

```

Create steps to run the demo.

```

import { readFileSync } from "node:fs";
import { join } from "node:path";

import axios from "axios";

import {
  DescribeTargetGroupsCommand,

```

```
    DescribeTargetHealthCommand,  
    ElasticLoadBalancingV2Client,  
  } from "@aws-sdk/client-elastic-load-balancing-v2";  
import {  
  DescribeInstanceInformationCommand,  
  PutParameterCommand,  
  SSMClient,  
  SendCommandCommand,  
} from "@aws-sdk/client-ssm";  
import {  
  IAMClient,  
  CreatePolicyCommand,  
  CreateRoleCommand,  
  AttachRolePolicyCommand,  
  CreateInstanceProfileCommand,  
  AddRoleToInstanceProfileCommand,  
  waitUntilInstanceProfileExists,  
} from "@aws-sdk/client-iam";  
import {  
  AutoScalingClient,  
  DescribeAutoScalingGroupsCommand,  
  TerminateInstanceInAutoScalingGroupCommand,  
} from "@aws-sdk/client-auto-scaling";  
import {  
  DescribeIamInstanceProfileAssociationsCommand,  
  EC2Client,  
  RebootInstancesCommand,  
  ReplaceIamInstanceProfileAssociationCommand,  
} from "@aws-sdk/client-ec2";  
  
import {  
  ScenarioAction,  
  ScenarioInput,  
  ScenarioOutput,  
} from "@aws-doc-sdk-examples/lib/scenario/scenario.js";  
import { retry } from "@aws-doc-sdk-examples/lib/utils/util-timers.js";  
  
import { MESSAGES, NAMES, RESOURCES_PATH } from "./constants.js";  
import { findLoadBalancer } from "./shared.js";  
  
const getRecommendation = new ScenarioAction(  
  "getRecommendation",  
  async (state) => {  
    const loadBalancer = await findLoadBalancer(NAMES.loadBalancerName);
```

```

    if (loadBalancer) {
      state.loadBalancerDnsName = loadBalancer.DNSName;
      try {
        state.recommendation = (
          await axios.get(`http://${state.loadBalancerDnsName}`)
        ).data;
      } catch (e) {
        state.recommendation = e instanceof Error ? e.message : e;
      }
    } else {
      throw new Error(MESSAGES.demoFindLoadBalancerError);
    }
  },
);

const getRecommendationResult = new ScenarioOutput(
  "getRecommendationResult",
  (state) =>
    `Recommendation:\n${JSON.stringify(state.recommendation, null, 2)}`,
  { preformatted: true },
);

const getHealthCheck = new ScenarioAction("getHealthCheck", async (state) => {
  const client = new ElasticLoadBalancingV2Client({});
  const { TargetGroups } = await client.send(
    new DescribeTargetGroupsCommand({
      Names: [NAMES.loadBalancerTargetGroupName],
    }),
  );
);

const { TargetHealthDescriptions } = await client.send(
  new DescribeTargetHealthCommand({
    TargetGroupArn: TargetGroups[0].TargetGroupArn,
  }),
);
state.targetHealthDescriptions = TargetHealthDescriptions;
});

const getHealthCheckResult = new ScenarioOutput(
  "getHealthCheckResult",
  /**
   * @param {{ targetHealthDescriptions: import('@aws-sdk/client-elastic-load-
   balancing-v2').TargetHealthDescription[]}} state
   */

```

```
(state) => {
  const status = state.targetHealthDescriptions
    .map((th) => `${th.Target.Id}: ${th.TargetHealth.State}`)
    .join("\n");
  return `Health check:\n${status}`;
},
{ preformatted: true },
);

const loadBalancerLoop = new ScenarioAction(
  "loadBalancerLoop",
  getRecommendation.action,
  {
    whileConfig: {
      whileFn: ({ loadBalancerCheck }) => loadBalancerCheck,
      input: new ScenarioInput(
        "loadBalancerCheck",
        MESSAGES.demoLoadBalancerCheck,
        {
          type: "confirm",
        },
      ),
      output: getRecommendationResult,
    },
  },
);

const healthCheckLoop = new ScenarioAction(
  "healthCheckLoop",
  getHealthCheck.action,
  {
    whileConfig: {
      whileFn: ({ healthCheck }) => healthCheck,
      input: new ScenarioInput("healthCheck", MESSAGES.demoHealthCheck, {
        type: "confirm",
      }),
      output: getHealthCheckResult,
    },
  },
);

const statusSteps = [
  getRecommendation,
  getRecommendationResult,
```

```
    getHealthCheck,
    getHealthCheckResult,
  ];

/**
 * @type {import('@aws-doc-sdk-examples/lib/scenario.js').Step[]}
 */
export const demoSteps = [
  new ScenarioOutput("header", MESSAGES.demoHeader, { header: true }),
  new ScenarioOutput("sanityCheck", MESSAGES.demoSanityCheck),
  ...statusSteps,
  new ScenarioInput(
    "brokenDependencyConfirmation",
    MESSAGES.demoBrokenDependencyConfirmation,
    { type: "confirm" },
  ),
  new ScenarioAction("brokenDependency", async (state) => {
    if (!state.brokenDependencyConfirmation) {
      process.exit();
    } else {
      const client = new SSMClient({});
      state.badTableName = `fake-table-${Date.now()}`;
      await client.send(
        new PutParameterCommand({
          Name: NAMES.ssmTableNameKey,
          Value: state.badTableName,
          Overwrite: true,
          Type: "String",
        }),
      );
    }
  }),
  new ScenarioOutput("testBrokenDependency", (state) =>
    MESSAGES.demoTestBrokenDependency.replace(
      "${TABLE_NAME}",
      state.badTableName,
    ),
  ),
  ...statusSteps,
  new ScenarioInput(
    "staticResponseConfirmation",
    MESSAGES.demoStaticResponseConfirmation,
    { type: "confirm" },
  ),
];
```

```
new ScenarioAction("staticResponse", async (state) => {
  if (!state.staticResponseConfirmation) {
    process.exit();
  } else {
    const client = new SSMClient({});
    await client.send(
      new PutParameterCommand({
        Name: NAMES.ssmFailureResponseKey,
        Value: "static",
        Overwrite: true,
        Type: "String",
      }),
    );
  }
}),
new ScenarioOutput("testStaticResponse", MESSAGES.demoTestStaticResponse),
...statusSteps,
new ScenarioInput(
  "badCredentialsConfirmation",
  MESSAGES.demoBadCredentialsConfirmation,
  { type: "confirm" },
),
new ScenarioAction("badCredentialsExit", (state) => {
  if (!state.badCredentialsConfirmation) {
    process.exit();
  }
}),
new ScenarioAction("fixDynamoDBName", async () => {
  const client = new SSMClient({});
  await client.send(
    new PutParameterCommand({
      Name: NAMES.ssmTableNameKey,
      Value: NAMES.tableName,
      Overwrite: true,
      Type: "String",
    }),
  );
}),
new ScenarioAction(
  "badCredentials",
  /**
   * @param {{ targetInstance: import('@aws-sdk/client-auto-scaling').Instance }}
state
  */
```



```
async (state) => {
  await createSsmOnlyInstanceProfile();
  const autoScalingClient = new AutoScalingClient({});
  const { AutoScalingGroups } = await autoScalingClient.send(
    new DescribeAutoScalingGroupsCommand({
      AutoScalingGroupNames: [NAMES.autoScalingGroupName],
    }),
  );
  state.targetInstance = AutoScalingGroups[0].Instances[0];
  const ec2Client = new EC2Client({});
  const { IamInstanceProfileAssociations } = await ec2Client.send(
    new DescribeIamInstanceProfileAssociationsCommand({
      Filters: [
        { Name: "instance-id", Values: [state.targetInstance.InstanceId] },
      ],
    }),
  );
  state.instanceProfileAssociationId =
    IamInstanceProfileAssociations[0].AssociationId;
  await retry({ intervalInMs: 1000, maxRetries: 30 }, () =>
    ec2Client.send(
      new ReplaceIamInstanceProfileAssociationCommand({
        AssociationId: state.instanceProfileAssociationId,
        IamInstanceProfile: { Name: NAMES.ssmOnlyInstanceProfileName },
      }),
    ),
  );

  await ec2Client.send(
    new RebootInstancesCommand({
      InstanceIds: [state.targetInstance.InstanceId],
    }),
  );

  const ssmClient = new SSMClient({});
  await retry({ intervalInMs: 20000, maxRetries: 15 }, async () => {
    const { InstanceInformationList } = await ssmClient.send(
      new DescribeInstanceInformationCommand({}),
    );

    const instance = InstanceInformationList.find(
      (info) => info.InstanceId === state.targetInstance.InstanceId,
    );
  });
}
```

```

        if (!instance) {
            throw new Error("Instance not found.");
        }
    });

    await ssmClient.send(
        new SendCommandCommand({
            InstanceIds: [state.targetInstance.InstanceId],
            DocumentName: "AWS-RunShellScript",
            Parameters: { commands: ["cd / && sudo python3 server.py 80"] },
        }),
    );
},
),
new ScenarioOutput(
    "testBadCredentials",
    /**
     * @param {{ targetInstance: import('@aws-sdk/client-ssm').InstanceInformation}}
state
    */
    (state) =>
        MESSAGES.demoTestBadCredentials.replace(
            "${INSTANCE_ID}",
            state.targetInstance.InstanceId,
        ),
),
loadBalancerLoop,
new ScenarioInput(
    "deepHealthCheckConfirmation",
    MESSAGES.demoDeepHealthCheckConfirmation,
    { type: "confirm" },
),
new ScenarioAction("deepHealthCheckExit", (state) => {
    if (!state.deepHealthCheckConfirmation) {
        process.exit();
    }
}),
new ScenarioAction("deepHealthCheck", async () => {
    const client = new SSMClient({});
    await client.send(
        new PutParameterCommand({
            Name: NAMES.ssmHealthCheckKey,
            Value: "deep",
            Overwrite: true,

```

```
        Type: "String",
      }),
    );
  })),
  new ScenarioOutput("testDeepHealthCheck", MESSAGES.demoTestDeepHealthCheck),
  healthCheckLoop,
  loadBalancerLoop,
  new ScenarioInput(
    "killInstanceConfirmation",
    /**
     * @param {{ targetInstance: import('@aws-sdk/client-
    ssm').InstanceInformation }} state
     */
    (state) =>
      MESSAGES.demoKillInstanceConfirmation.replace(
        "${INSTANCE_ID}",
        state.targetInstance.InstanceId,
      ),
    { type: "confirm" },
  ),
  new ScenarioAction("killInstanceExit", (state) => {
    if (!state.killInstanceConfirmation) {
      process.exit();
    }
  })),
  new ScenarioAction(
    "killInstance",
    /**
     * @param {{ targetInstance: import('@aws-sdk/client-
    ssm').InstanceInformation }} state
     */
    async (state) => {
      const client = new AutoScalingClient({});
      await client.send(
        new TerminateInstanceInAutoScalingGroupCommand({
          InstanceId: state.targetInstance.InstanceId,
          ShouldDecrementDesiredCapacity: false,
        }),
      );
    },
  ),
  new ScenarioOutput("testKillInstance", MESSAGES.demoTestKillInstance),
  healthCheckLoop,
  loadBalancerLoop,
```

```
new ScenarioInput("failOpenConfirmation", MESSAGES.demoFailOpenConfirmation, {
  type: "confirm",
}),
new ScenarioAction("failOpenExit", (state) => {
  if (!state.failOpenConfirmation) {
    process.exit();
  }
}),
new ScenarioAction("failOpen", () => {
  const client = new SSMClient({});
  return client.send(
    new PutParameterCommand({
      Name: NAMES.ssmTableNameKey,
      Value: `fake-table-${Date.now()}`,
      Overwrite: true,
      Type: "String",
    }),
  );
}),
new ScenarioOutput("testFailOpen", MESSAGES.demoFailOpenTest),
healthCheckLoop,
loadBalancerLoop,
new ScenarioInput(
  "resetTableConfirmation",
  MESSAGES.demoResetTableConfirmation,
  { type: "confirm" },
),
new ScenarioAction("resetTableExit", (state) => {
  if (!state.resetTableConfirmation) {
    process.exit();
  }
}),
new ScenarioAction("resetTable", async () => {
  const client = new SSMClient({});
  await client.send(
    new PutParameterCommand({
      Name: NAMES.ssmTableNameKey,
      Value: NAMES.tableName,
      Overwrite: true,
      Type: "String",
    }),
  );
}),
new ScenarioOutput("testResetTable", MESSAGES.demoTestResetTable),
```

```
healthCheckLoop,
loadBalancerLoop,
];

async function createSsmOnlyInstanceProfile() {
  const iamClient = new IAMClient({});
  const { Policy } = await iamClient.send(
    new CreatePolicyCommand({
      PolicyName: NAMES.ssmOnlyPolicyName,
      PolicyDocument: readFileSync(
        join(RESOURCES_PATH, "ssm_only_policy.json"),
      ),
    }),
  );
  await iamClient.send(
    new CreateRoleCommand({
      RoleName: NAMES.ssmOnlyRoleName,
      AssumeRolePolicyDocument: JSON.stringify({
        Version: "2012-10-17",
        Statement: [
          {
            Effect: "Allow",
            Principal: { Service: "ec2.amazonaws.com" },
            Action: "sts:AssumeRole",
          },
        ],
      }),
    ));
  await iamClient.send(
    new AttachRolePolicyCommand({
      RoleName: NAMES.ssmOnlyRoleName,
      PolicyArn: Policy.Arn,
    }),
  );
  await iamClient.send(
    new AttachRolePolicyCommand({
      RoleName: NAMES.ssmOnlyRoleName,
      PolicyArn: "arn:aws:iam::aws:policy/AmazonSSMManagedInstanceCore",
    }),
  );
  const { InstanceProfile } = await iamClient.send(
    new CreateInstanceProfileCommand({
      InstanceProfileName: NAMES.ssmOnlyInstanceProfileName,
```

```
    }),
  );
  await waitUntilInstanceProfileExists(
    { client: iamClient },
    { InstanceProfileName: NAMES.ssmOnlyInstanceProfileName },
  );
  await iamClient.send(
    new AddRoleToInstanceProfileCommand({
      InstanceProfileName: NAMES.ssmOnlyInstanceProfileName,
      RoleName: NAMES.ssmOnlyRoleName,
    })),
  );

  return InstanceProfile;
}
```

Create steps to destroy all of the resources.

```
import { unlinkSync } from "node:fs";

import { DynamoDBClient, DeleteTableCommand } from "@aws-sdk/client-dynamodb";
import {
  EC2Client,
  DeleteKeyPairCommand,
  DeleteLaunchTemplateCommand,
  RevokeSecurityGroupIngressCommand,
} from "@aws-sdk/client-ec2";
import {
  IAMClient,
  DeleteInstanceProfileCommand,
  RemoveRoleFromInstanceProfileCommand,
  DeletePolicyCommand,
  DeleteRoleCommand,
  DetachRolePolicyCommand,
  paginateListPolicies,
} from "@aws-sdk/client-iam";
import {
  AutoScalingClient,
  DeleteAutoScalingGroupCommand,
  TerminateInstanceInAutoScalingGroupCommand,
  UpdateAutoScalingGroupCommand,
  paginateDescribeAutoScalingGroups,
```

```

} from "@aws-sdk/client-auto-scaling";
import {
  DeleteLoadBalancerCommand,
  DeleteTargetGroupCommand,
  DescribeTargetGroupsCommand,
  ElasticLoadBalancingV2Client,
} from "@aws-sdk/client-elastic-load-balancing-v2";

import {
  ScenarioOutput,
  ScenarioInput,
  ScenarioAction,
} from "@aws-doc-sdk-examples/lib/scenario/index.js";
import { loadState } from "@aws-doc-sdk-examples/lib/scenario/steps-common.js";
import { retry } from "@aws-doc-sdk-examples/lib/utils/util-timers.js";

import { MESSAGES, NAMES } from "./constants.js";
import { findLoadBalancer } from "./shared.js";

/**
 * @type {import('@aws-doc-sdk-examples/lib/scenario.js').Step[]}
 */
export const destroySteps = [
  loadState,
  new ScenarioInput("destroy", MESSAGES.destroy, { type: "confirm" }),
  new ScenarioAction(
    "abort",
    (state) => state.destroy === false && process.exit(),
  ),
  new ScenarioAction("deleteTable", async (c) => {
    try {
      const client = new DynamoDBClient({});
      await client.send(new DeleteTableCommand({ TableName: NAMES.tableName }));
    } catch (e) {
      c.deleteTableError = e;
    }
  }),
  new ScenarioOutput("deleteTableResult", (state) => {
    if (state.deleteTableError) {
      console.error(state.deleteTableError);
      return MESSAGES.deleteTableError.replace(
        "${TABLE_NAME}",
        NAMES.tableName,
      );
    }
  });

```

```
    } else {
      return MESSAGES.deletedTable.replace("${TABLE_NAME}", NAMES.tableName);
    }
  })),
  new ScenarioAction("deleteKeyPair", async (state) => {
    try {
      const client = new EC2Client({});
      await client.send(
        new DeleteKeyPairCommand({ KeyName: NAMES.keyPairName }),
      );
      unlinkSync(`${NAMES.keyPairName}.pem`);
    } catch (e) {
      state.deleteKeyPairError = e;
    }
  })),
  new ScenarioOutput("deleteKeyPairResult", (state) => {
    if (state.deleteKeyPairError) {
      console.error(state.deleteKeyPairError);
      return MESSAGES.deleteKeyPairError.replace(
        "${KEY_PAIR_NAME}",
        NAMES.keyPairName,
      );
    } else {
      return MESSAGES.deletedKeyPair.replace(
        "${KEY_PAIR_NAME}",
        NAMES.keyPairName,
      );
    }
  })),
  new ScenarioAction("detachPolicyFromRole", async (state) => {
    try {
      const client = new IAMClient({});
      const policy = await findPolicy(NAMES.instancePolicyName);

      if (!policy) {
        state.detachPolicyFromRoleError = new Error(
          `Policy ${NAMES.instancePolicyName} not found.`
        );
      } else {
        await client.send(
          new DetachRolePolicyCommand({
            RoleName: NAMES.instanceRoleName,
            PolicyArn: policy.Arn,
          })
        );
      }
    }
  })
}
```



```
    );
  }
} catch (e) {
  state.detachPolicyFromRoleError = e;
}
}),
new ScenarioOutput("detachedPolicyFromRole", (state) => {
  if (state.detachPolicyFromRoleError) {
    console.error(state.detachPolicyFromRoleError);
    return MESSAGES.detachPolicyFromRoleError
      .replace("${INSTANCE_POLICY_NAME}", NAMES.instancePolicyName)
      .replace("${INSTANCE_ROLE_NAME}", NAMES.instanceRoleName);
  } else {
    return MESSAGES.detachedPolicyFromRole
      .replace("${INSTANCE_POLICY_NAME}", NAMES.instancePolicyName)
      .replace("${INSTANCE_ROLE_NAME}", NAMES.instanceRoleName);
  }
}),
new ScenarioAction("deleteInstancePolicy", async (state) => {
  const client = new IAMClient({});
  const policy = await findPolicy(NAMES.instancePolicyName);

  if (!policy) {
    state.deletePolicyError = new Error(
      `Policy ${NAMES.instancePolicyName} not found.`
    );
  } else {
    return client.send(
      new DeletePolicyCommand({
        PolicyArn: policy.Arn,
      }),
    );
  }
}),
new ScenarioOutput("deletePolicyResult", (state) => {
  if (state.deletePolicyError) {
    console.error(state.deletePolicyError);
    return MESSAGES.deletePolicyError.replace(
      "${INSTANCE_POLICY_NAME}",
      NAMES.instancePolicyName,
    );
  } else {
    return MESSAGES.deletedPolicy.replace(
      "${INSTANCE_POLICY_NAME}",
```

```
        NAMES.instancePolicyName,
    );
  }
}),
new ScenarioAction("removeRoleFromInstanceProfile", async (state) => {
  try {
    const client = new IAMClient({});
    await client.send(
      new RemoveRoleFromInstanceProfileCommand({
        RoleName: NAMES.instanceRoleName,
        InstanceProfileName: NAMES.instanceProfileName,
      }),
    );
  } catch (e) {
    state.removeRoleFromInstanceProfileError = e;
  }
}),
new ScenarioOutput("removeRoleFromInstanceProfileResult", (state) => {
  if (state.removeRoleFromInstanceProfile) {
    console.error(state.removeRoleFromInstanceProfileError);
    return MESSAGES.removeRoleFromInstanceProfileError
      .replace("${INSTANCE_PROFILE_NAME}", NAMES.instanceProfileName)
      .replace("${INSTANCE_ROLE_NAME}", NAMES.instanceRoleName);
  } else {
    return MESSAGES.removedRoleFromInstanceProfile
      .replace("${INSTANCE_PROFILE_NAME}", NAMES.instanceProfileName)
      .replace("${INSTANCE_ROLE_NAME}", NAMES.instanceRoleName);
  }
}),
new ScenarioAction("deleteInstanceRole", async (state) => {
  try {
    const client = new IAMClient({});
    await client.send(
      new DeleteRoleCommand({
        RoleName: NAMES.instanceRoleName,
      }),
    );
  } catch (e) {
    state.deleteInstanceRoleError = e;
  }
}),
new ScenarioOutput("deleteInstanceRoleResult", (state) => {
  if (state.deleteInstanceRoleError) {
    console.error(state.deleteInstanceRoleError);
  }
});
```

```
    return MESSAGES.deleteInstanceRoleError.replace(
      "${INSTANCE_ROLE_NAME}",
      NAMES.instanceRoleName,
    );
  } else {
    return MESSAGES.deletedInstanceRole.replace(
      "${INSTANCE_ROLE_NAME}",
      NAMES.instanceRoleName,
    );
  }
}),
new ScenarioAction("deleteInstanceProfile", async (state) => {
  try {
    const client = new IAMClient({});
    await client.send(
      new DeleteInstanceProfileCommand({
        InstanceProfileName: NAMES.instanceProfileName,
      }),
    );
  } catch (e) {
    state.deleteInstanceProfileError = e;
  }
}),
new ScenarioOutput("deleteInstanceProfileResult", (state) => {
  if (state.deleteInstanceProfileError) {
    console.error(state.deleteInstanceProfileError);
    return MESSAGES.deleteInstanceProfileError.replace(
      "${INSTANCE_PROFILE_NAME}",
      NAMES.instanceProfileName,
    );
  } else {
    return MESSAGES.deletedInstanceProfile.replace(
      "${INSTANCE_PROFILE_NAME}",
      NAMES.instanceProfileName,
    );
  }
}),
new ScenarioAction("deleteAutoScalingGroup", async (state) => {
  try {
    await terminateGroupInstances(NAMES.autoScalingGroupName);
    await retry({ intervalInMs: 60000, maxRetries: 60 }, async () => {
      await deleteAutoScalingGroup(NAMES.autoScalingGroupName);
    });
  } catch (e) {
```

```
    state.deleteAutoScalingGroupError = e;
  }
}),
new ScenarioOutput("deleteAutoScalingGroupResult", (state) => {
  if (state.deleteAutoScalingGroupError) {
    console.error(state.deleteAutoScalingGroupError);
    return MESSAGES.deleteAutoScalingGroupError.replace(
      "${AUTO_SCALING_GROUP_NAME}",
      NAMES.autoScalingGroupName,
    );
  } else {
    return MESSAGES.deletedAutoScalingGroup.replace(
      "${AUTO_SCALING_GROUP_NAME}",
      NAMES.autoScalingGroupName,
    );
  }
}),
new ScenarioAction("deleteLaunchTemplate", async (state) => {
  const client = new EC2Client({});
  try {
    await client.send(
      new DeleteLaunchTemplateCommand({
        LaunchTemplateName: NAMES.launchTemplateName,
      }),
    );
  } catch (e) {
    state.deleteLaunchTemplateError = e;
  }
}),
new ScenarioOutput("deleteLaunchTemplateResult", (state) => {
  if (state.deleteLaunchTemplateError) {
    console.error(state.deleteLaunchTemplateError);
    return MESSAGES.deleteLaunchTemplateError.replace(
      "${LAUNCH_TEMPLATE_NAME}",
      NAMES.launchTemplateName,
    );
  } else {
    return MESSAGES.deletedLaunchTemplate.replace(
      "${LAUNCH_TEMPLATE_NAME}",
      NAMES.launchTemplateName,
    );
  }
}),
new ScenarioAction("deleteLoadBalancer", async (state) => {
```

```
    try {
      const client = new ElasticLoadBalancingV2Client({});
      const loadBalancer = await findLoadBalancer(NAMES.loadBalancerName);
      await client.send(
        new DeleteLoadBalancerCommand({
          LoadBalancerArn: loadBalancer.LoadBalancerArn,
        }),
      );
      await retry({ intervalInMs: 1000, maxRetries: 60 }, async () => {
        const lb = await findLoadBalancer(NAMES.loadBalancerName);
        if (lb) {
          throw new Error("Load balancer still exists.");
        }
      });
    } catch (e) {
      state.deleteLoadBalancerError = e;
    }
  })),
  new ScenarioOutput("deleteLoadBalancerResult", (state) => {
    if (state.deleteLoadBalancerError) {
      console.error(state.deleteLoadBalancerError);
      return MESSAGES.deleteLoadBalancerError.replace(
        "${LB_NAME}",
        NAMES.loadBalancerName,
      );
    } else {
      return MESSAGES.deletedLoadBalancer.replace(
        "${LB_NAME}",
        NAMES.loadBalancerName,
      );
    }
  })),
  new ScenarioAction("deleteLoadBalancerTargetGroup", async (state) => {
    const client = new ElasticLoadBalancingV2Client({});
    try {
      const { TargetGroups } = await client.send(
        new DescribeTargetGroupsCommand({
          Names: [NAMES.loadBalancerTargetGroupName],
        }),
      );
    }

    await retry({ intervalInMs: 1000, maxRetries: 30 }, () => {
      client.send(
        new DeleteTargetGroupCommand({
```

```
        TargetGroupArn: TargetGroups[0].TargetGroupArn,
      )),
    ),
  );
} catch (e) {
  state.deleteLoadBalancerTargetGroupError = e;
}
}),
new ScenarioOutput("deleteLoadBalancerTargetGroupResult", (state) => {
  if (state.deleteLoadBalancerTargetGroupError) {
    console.error(state.deleteLoadBalancerTargetGroupError);
    return MESSAGES.deleteLoadBalancerTargetGroupError.replace(
      "${TARGET_GROUP_NAME}",
      NAMES.loadBalancerTargetGroupName,
    );
  } else {
    return MESSAGES.deletedLoadBalancerTargetGroup.replace(
      "${TARGET_GROUP_NAME}",
      NAMES.loadBalancerTargetGroupName,
    );
  }
}),
new ScenarioAction("detachSsmOnlyRoleFromProfile", async (state) => {
  try {
    const client = new IAMClient({});
    await client.send(
      new RemoveRoleFromInstanceProfileCommand({
        InstanceProfileName: NAMES.ssmOnlyInstanceProfileName,
        RoleName: NAMES.ssmOnlyRoleName,
      })),
    );
  } catch (e) {
    state.detachSsmOnlyRoleFromProfileError = e;
  }
}),
new ScenarioOutput("detachSsmOnlyRoleFromProfileResult", (state) => {
  if (state.detachSsmOnlyRoleFromProfileError) {
    console.error(state.detachSsmOnlyRoleFromProfileError);
    return MESSAGES.detachSsmOnlyRoleFromProfileError
      .replace("${ROLE_NAME}", NAMES.ssmOnlyRoleName)
      .replace("${PROFILE_NAME}", NAMES.ssmOnlyInstanceProfileName);
  } else {
    return MESSAGES.detachedSsmOnlyRoleFromProfile
      .replace("${ROLE_NAME}", NAMES.ssmOnlyRoleName)
  }
})
```

```
        .replace("${PROFILE_NAME}", NAMES.ssmOnlyInstanceProfileName);
    }
  })),
  new ScenarioAction("detachSsmOnlyCustomRolePolicy", async (state) => {
    try {
      const iamClient = new IAMClient({});
      const ssmOnlyPolicy = await findPolicy(NAMES.ssmOnlyPolicyName);
      await iamClient.send(
        new DetachRolePolicyCommand({
          RoleName: NAMES.ssmOnlyRoleName,
          PolicyArn: ssmOnlyPolicy.Arn,
        })),
      );
    } catch (e) {
      state.detachSsmOnlyCustomRolePolicyError = e;
    }
  })),
  new ScenarioOutput("detachSsmOnlyCustomRolePolicyResult", (state) => {
    if (state.detachSsmOnlyCustomRolePolicyError) {
      console.error(state.detachSsmOnlyCustomRolePolicyError);
      return MESSAGES.detachSsmOnlyCustomRolePolicyError
        .replace("${ROLE_NAME}", NAMES.ssmOnlyRoleName)
        .replace("${POLICY_NAME}", NAMES.ssmOnlyPolicyName);
    } else {
      return MESSAGES.detachedSsmOnlyCustomRolePolicy
        .replace("${ROLE_NAME}", NAMES.ssmOnlyRoleName)
        .replace("${POLICY_NAME}", NAMES.ssmOnlyPolicyName);
    }
  })),
  new ScenarioAction("detachSsmOnlyAWSRolePolicy", async (state) => {
    try {
      const iamClient = new IAMClient({});
      await iamClient.send(
        new DetachRolePolicyCommand({
          RoleName: NAMES.ssmOnlyRoleName,
          PolicyArn: "arn:aws:iam::aws:policy/AmazonSSMManagedInstanceCore",
        })),
      );
    } catch (e) {
      state.detachSsmOnlyAWSRolePolicyError = e;
    }
  })),
  new ScenarioOutput("detachSsmOnlyAWSRolePolicyResult", (state) => {
    if (state.detachSsmOnlyAWSRolePolicyError) {
```

```

    console.error(state.detachSsmOnlyAWSRolePolicyError);
    return MESSAGES.detachSsmOnlyAWSRolePolicyError
      .replace("${ROLE_NAME}", NAMES.ssmOnlyRoleName)
      .replace("${POLICY_NAME}", "AmazonSSManagedInstanceCore");
  } else {
    return MESSAGES.detachedSsmOnlyAWSRolePolicy
      .replace("${ROLE_NAME}", NAMES.ssmOnlyRoleName)
      .replace("${POLICY_NAME}", "AmazonSSManagedInstanceCore");
  }
}),
new ScenarioAction("deleteSsmOnlyInstanceProfile", async (state) => {
  try {
    const iamClient = new IAMClient({});
    await iamClient.send(
      new DeleteInstanceProfileCommand({
        InstanceProfileName: NAMES.ssmOnlyInstanceProfileName,
      }),
    );
  } catch (e) {
    state.deleteSsmOnlyInstanceProfileError = e;
  }
}),
new ScenarioOutput("deleteSsmOnlyInstanceProfileResult", (state) => {
  if (state.deleteSsmOnlyInstanceProfileError) {
    console.error(state.deleteSsmOnlyInstanceProfileError);
    return MESSAGES.deleteSsmOnlyInstanceProfileError.replace(
      "${INSTANCE_PROFILE_NAME}",
      NAMES.ssmOnlyInstanceProfileName,
    );
  } else {
    return MESSAGES.deletedSsmOnlyInstanceProfile.replace(
      "${INSTANCE_PROFILE_NAME}",
      NAMES.ssmOnlyInstanceProfileName,
    );
  }
}),
new ScenarioAction("deleteSsmOnlyPolicy", async (state) => {
  try {
    const iamClient = new IAMClient({});
    const ssmOnlyPolicy = await findPolicy(NAMES.ssmOnlyPolicyName);
    await iamClient.send(
      new DeletePolicyCommand({
        PolicyArn: ssmOnlyPolicy.Arn,
      }),
    );
  }
});

```



```
    );
  } catch (e) {
    state.deleteSsmOnlyPolicyError = e;
  }
}),
new ScenarioOutput("deleteSsmOnlyPolicyResult", (state) => {
  if (state.deleteSsmOnlyPolicyError) {
    console.error(state.deleteSsmOnlyPolicyError);
    return MESSAGES.deleteSsmOnlyPolicyError.replace(
      "${POLICY_NAME}",
      NAMES.ssmOnlyPolicyName,
    );
  } else {
    return MESSAGES.deletedSsmOnlyPolicy.replace(
      "${POLICY_NAME}",
      NAMES.ssmOnlyPolicyName,
    );
  }
}),
new ScenarioAction("deleteSsmOnlyRole", async (state) => {
  try {
    const iamClient = new IAMClient({});
    await iamClient.send(
      new DeleteRoleCommand({
        RoleName: NAMES.ssmOnlyRoleName,
      }),
    );
  } catch (e) {
    state.deleteSsmOnlyRoleError = e;
  }
}),
new ScenarioOutput("deleteSsmOnlyRoleResult", (state) => {
  if (state.deleteSsmOnlyRoleError) {
    console.error(state.deleteSsmOnlyRoleError);
    return MESSAGES.deleteSsmOnlyRoleError.replace(
      "${ROLE_NAME}",
      NAMES.ssmOnlyRoleName,
    );
  } else {
    return MESSAGES.deletedSsmOnlyRole.replace(
      "${ROLE_NAME}",
      NAMES.ssmOnlyRoleName,
    );
  }
}
```

```

    })),
    new ScenarioAction(
      "revokeSecurityGroupIngress",
      async (
        /** @type {{ myIp: string, defaultSecurityGroup: { GroupId: string } }} */
state,
      ) => {
        const ec2Client = new EC2Client({});

        try {
          await ec2Client.send(
            new RevokeSecurityGroupIngressCommand({
              GroupId: state.defaultSecurityGroup.GroupId,
              CidrIp: `${state.myIp}/32`,
              FromPort: 80,
              ToPort: 80,
              IpProtocol: "tcp",
            }),
          );
        } catch (e) {
          state.revokeSecurityGroupIngressError = e;
        }
      },
    ),
    new ScenarioOutput("revokeSecurityGroupIngressResult", (state) => {
      if (state.revokeSecurityGroupIngressError) {
        console.error(state.revokeSecurityGroupIngressError);
        return MESSAGES.revokeSecurityGroupIngressError.replace(
          "${IP}",
          state.myIp,
        );
      } else {
        return MESSAGES.revokedSecurityGroupIngress.replace("${IP}", state.myIp);
      }
    }),
  ],
);

/**
 * @param {string} policyName
 */
async function findPolicy(policyName) {
  const client = new IAMClient({});
  const paginatedPolicies = paginateListPolicies({ client }, {});
  for await (const page of paginatedPolicies) {

```

```
    const policy = page.Policies.find((p) => p.PolicyName === policyName);
    if (policy) {
      return policy;
    }
  }
}

/**
 * @param {string} groupName
 */
async function deleteAutoScalingGroup(groupName) {
  const client = new AutoScalingClient({});
  try {
    await client.send(
      new DeleteAutoScalingGroupCommand({
        AutoScalingGroupName: groupName,
      }),
    );
  } catch (err) {
    if (!(err instanceof Error)) {
      throw err;
    } else {
      console.log(err.name);
      throw err;
    }
  }
}

/**
 * @param {string} groupName
 */
async function terminateGroupInstances(groupName) {
  const autoScalingClient = new AutoScalingClient({});
  const group = await findAutoScalingGroup(groupName);
  await autoScalingClient.send(
    new UpdateAutoScalingGroupCommand({
      AutoScalingGroupName: group.AutoScalingGroupName,
      MinSize: 0,
    }),
  );
  for (const i of group.Instances) {
    await retry({ intervalInMs: 1000, maxRetries: 30 }, () =>
      autoScalingClient.send(
        new TerminateInstanceInAutoScalingGroupCommand({
```

```
        InstanceId: i.InstanceId,
        ShouldDecrementDesiredCapacity: true,
      })),
    ),
  );
}
}

async function findAutoScalingGroup(groupName) {
  const client = new AutoScalingClient({});
  const paginatedGroups = paginateDescribeAutoScalingGroups({ client }, {});
  for await (const page of paginatedGroups) {
    const group = page.AutoScalingGroups.find(
      (g) => g.AutoScalingGroupName === groupName,
    );
    if (group) {
      return group;
    }
  }
  throw new Error(`Auto scaling group ${groupName} not found.`);
}
```

- For API details, see the following topics in *AWS SDK for JavaScript API Reference*.
 - [AttachLoadBalancerTargetGroups](#)
 - [CreateAutoScalingGroup](#)
 - [CreateInstanceProfile](#)
 - [CreateLaunchTemplate](#)
 - [CreateListener](#)
 - [CreateLoadBalancer](#)
 - [CreateTargetGroup](#)
 - [DeleteAutoScalingGroup](#)
 - [DeleteInstanceProfile](#)
 - [DeleteLaunchTemplate](#)
 - [DeleteLoadBalancer](#)
 - [DeleteTargetGroup](#)
 - [DescribeAutoScalingGroups](#)

- [DescribeAvailabilityZones](#)
- [DescribeIamInstanceProfileAssociations](#)
- [DescribeInstances](#)
- [DescribeLoadBalancers](#)
- [DescribeSubnets](#)
- [DescribeTargetGroups](#)
- [DescribeTargetHealth](#)
- [DescribeVpcs](#)
- [RebootInstances](#)
- [ReplacelamInstanceProfileAssociation](#)
- [TerminateInstanceInAutoScalingGroup](#)
- [UpdateAutoScalingGroup](#)

Amazon Bedrock examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with Amazon Bedrock.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Get started

Hello Amazon Bedrock

The following code examples show how to get started using Amazon Bedrock.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { fileURLToPath } from "url";

import {
  BedrockClient,
  ListFoundationModelsCommand,
} from "@aws-sdk/client-bedrock";

const REGION = "us-east-1";
const client = new BedrockClient({ region: REGION });

export const main = async () => {
  const command = new ListFoundationModelsCommand({});

  const response = await client.send(command);
  const models = response.modelSummaries;

  console.log("Listing the available Bedrock foundation models:");

  for (let model of models) {
    console.log("=".repeat(42));
    console.log(` Model: ${model.modelId}`);
    console.log("-".repeat(42));
    console.log(` Name: ${model.modelName}`);
    console.log(` Provider: ${model.providerName}`);
    console.log(` Model ARN: ${model.modelArn}`);
    console.log(` Input modalities: ${model.inputModalities}`);
    console.log(` Output modalities: ${model.outputModalities}`);
    console.log(` Supported customizations: ${model.customizationsSupported}`);
    console.log(` Supported inference types: ${model.inferenceTypesSupported}`);
    console.log(` Lifecycle status: ${model.modelLifecycle.status}`);
    console.log("=".repeat(42) + "\n");
  }

  const active = models.filter(
    (m) => m.modelLifecycle.status === "ACTIVE",
  ).length;
  const legacy = models.filter(
    (m) => m.modelLifecycle.status === "LEGACY",
  ).length;

  console.log(
```

```
    `There are ${active} active and ${legacy} legacy foundation models in
    ${REGION}.`,
    );

    return response;
  };

  // Invoke main function if this file was run directly.
  if (process.argv[1] === fileURLToPath(import.meta.url)) {
    await main();
  }
}
```

- For API details, see [ListFoundationModels](#) in *AWS SDK for JavaScript API Reference*.

Topics

- [Actions](#)

Actions

GetFoundationModel

The following code example shows how to use `GetFoundationModel`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Get details about a foundation model.

```
import { fileURLToPath } from "url";

import {
  BedrockClient,
  GetFoundationModelCommand,
```

```
} from "@aws-sdk/client-bedrock";

/**
 * Get details about an Amazon Bedrock foundation model.
 *
 * @return {FoundationModelDetails} - The list of available bedrock foundation
 * models.
 */
export const getFoundationModel = async () => {
  const client = new BedrockClient();

  const command = new GetFoundationModelCommand({
    modelIdentifier: "amazon.titan-embed-text-v1",
  });

  const response = await client.send(command);

  return response.modelDetails;
};

// Invoke main function if this file was run directly.
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  const model = await getFoundationModel();
  console.log(model);
}
```

- For API details, see [GetFoundationModel](#) in *AWS SDK for JavaScript API Reference*.

ListFoundationModels

The following code example shows how to use ListFoundationModels.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

List the available foundation models.


```
import { fileURLToPath } from "url";

import {
  BedrockClient,
  ListFoundationModelsCommand,
} from "@aws-sdk/client-bedrock";

/**
 * List the available Amazon Bedrock foundation models.
 *
 * @return {FoundationModelSummary[]} - The list of available bedrock foundation
 * models.
 */
export const listFoundationModels = async () => {
  const client = new BedrockClient();

  const input = {
    // byProvider: 'STRING_VALUE',
    // byCustomizationType: 'FINE_TUNING' || 'CONTINUED_PRE_TRAINING',
    // byOutputModality: 'TEXT' || 'IMAGE' || 'EMBEDDING',
    // byInferenceType: 'ON_DEMAND' || 'PROVISIONED',
  };

  const command = new ListFoundationModelsCommand(input);

  const response = await client.send(command);

  return response.modelSummaries;
};

// Invoke main function if this file was run directly.
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  const models = await listFoundationModels();
  console.log(models);
}
```

- For API details, see [ListFoundationModels](#) in *AWS SDK for JavaScript API Reference*.

Amazon Bedrock Runtime examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with Amazon Bedrock Runtime.

Scenarios are code examples that show you how to accomplish specific tasks by calling multiple functions within a service or combined with other AWS services.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Get started

Hello Amazon Bedrock

The following code examples show how to get started using Amazon Bedrock.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
/**
 * @typedef {Object} Content
 * @property {string} text
 *
 * @typedef {Object} Usage
 * @property {number} input_tokens
 * @property {number} output_tokens
 *
 * @typedef {Object} ResponseBody
 * @property {Content[]} content
 * @property {Usage} usage
 */

import { fileURLToPath } from "url";
```

```
import {
  BedrockRuntimeClient,
  InvokeModelCommand,
} from "@aws-sdk/client-bedrock-runtime";

const AWS_REGION = "us-east-1";

const MODEL_ID = "anthropic.claude-3-haiku-20240307-v1:0";
const PROMPT = "Hi. In a short paragraph, explain what you can do.";

const hello = async () => {
  console.log("=".repeat(35));
  console.log("Welcome to the Amazon Bedrock demo!");
  console.log("=".repeat(35));

  console.log("Model: Anthropic Claude 3 Haiku");
  console.log(`Prompt: ${PROMPT}\n`);
  console.log("Invoking model...\n");

  // Create a new Bedrock Runtime client instance.
  const client = new BedrockRuntimeClient({ region: AWS_REGION });

  // Prepare the payload for the model.
  const payload = {
    anthropic_version: "bedrock-2023-05-31",
    max_tokens: 1000,
    messages: [{ role: "user", content: [{ type: "text", text: PROMPT }] }],
  };

  // Invoke Claude with the payload and wait for the response.
  const apiResponse = await client.send(
    new InvokeModelCommand({
      contentType: "application/json",
      body: JSON.stringify(payload),
      modelId: MODEL_ID,
    }),
  );

  // Decode and return the response(s)
  const decodedResponseBody = new TextDecoder().decode(apiResponse.body);
  /** @type {ResponseBody} */
  const responseBody = JSON.parse(decodedResponseBody);
  const responses = responseBody.content;
```

```
if (responses.length === 1) {
  console.log(`Response: ${responses[0].text}`);
} else {
  console.log("Haiku returned multiple responses:");
  console.log(responses);
}

console.log(`\nNumber of input tokens:  ${responseBody.usage.input_tokens}`);
console.log(`Number of output tokens:  ${responseBody.usage.output_tokens}`);
};

if (process.argv[1] === fileURLToPath(import.meta.url)) {
  await hello();
}
```

- For API details, see [InvokeModel](#) in *AWS SDK for JavaScript API Reference*.

Topics

- [Scenarios](#)
- [AI21 Labs Jurassic-2](#)
- [Amazon Titan Text](#)
- [Anthropic Claude](#)
- [Cohere Command](#)
- [Meta Llama](#)
- [Mistral AI](#)

Scenarios

Invoke multiple foundation models on Amazon Bedrock

The following code example shows how to prepare and send a prompt to a variety of large-language models (LLMs) on Amazon Bedrock

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { fileURLToPath } from "url";
import {
  Scenario,
  ScenarioAction,
  ScenarioInput,
  ScenarioOutput,
} from "@aws-doc-sdk-examples/lib/scenario/index.js";
import { FoundationModels } from "../config/foundation_models.js";

/**
 * @typedef {Object} ModelConfig
 * @property {Function} module
 * @property {Function} invoker
 * @property {string} modelId
 * @property {string} modelName
 */

const greeting = new ScenarioOutput(
  "greeting",
  "Welcome to the Amazon Bedrock Runtime client demo!",
  { header: true },
);

const selectModel = new ScenarioInput("model", "First, select a model:", {
  type: "select",
  choices: Object.values(FoundationModels).map((model) => ({
    name: model.modelName,
    value: model,
  })),
});

const enterPrompt = new ScenarioInput("prompt", "Now, enter your prompt:", {
  type: "input",
```

```
});

const printDetails = new ScenarioOutput(
  "print details",
  /**
   * @param {{ model: ModelConfig, prompt: string }} c
   */
  (c) => console.log(`Invoking ${c.model.modelName} with '${c.prompt}'...`),
  { slow: false },
);

const invokeModel = new ScenarioAction(
  "invoke model",
  /**
   * @param {{ model: ModelConfig, prompt: string, response: string }} c
   */
  async (c) => {
    const modelModule = await c.model.module();
    const invoker = c.model.invoker(modelModule);
    c.response = await invoker(c.prompt, c.model.modelId);
  },
);

const printResponse = new ScenarioOutput(
  "print response",
  /**
   * @param {{ response: string }} c
   */
  (c) => c.response,
  { slow: false },
);

const scenario = new Scenario("Amazon Bedrock Runtime Demo", [
  greeting,
  selectModel,
  enterPrompt,
  printDetails,
  invokeModel,
  printResponse,
]);

if (process.argv[1] === fileURLToPath(import.meta.url)) {
  scenario.run();
}
```

- For API details, see the following topics in *AWS SDK for JavaScript API Reference*.
 - [InvokeModel](#)
 - [InvokeModelWithResponseStream](#)

AI21 Labs Jurassic-2

Converse

The following code example shows how to send a text message to AI21 Labs Jurassic-2, using Bedrock's Converse API.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Send a text message to AI21 Labs Jurassic-2, using Bedrock's Converse API.

```
// Use the Conversation API to send a text message to AI21 Labs Jurassic-2.

import {
  BedrockRuntimeClient,
  ConverseCommand,
} from "@aws-sdk/client-bedrock-runtime";

// Create a Bedrock Runtime client in the AWS Region you want to use.
const client = new BedrockRuntimeClient({ region: "us-east-1" });

// Set the model ID, e.g., Jurassic-2 Mid.
const modelId = "ai21.j2-mid-v1";

// Start a conversation with the user message.
const userMessage =
  "Describe the purpose of a 'hello world' program in one line.";
const conversation = [
  {
```

```
    role: "user",
    content: [{ text: userMessage }],
  },
];

// Create a command with the model ID, the message, and a basic configuration.
const command = new ConverseCommand({
  modelId,
  messages: conversation,
  inferenceConfig: { maxTokens: 512, temperature: 0.5, topP: 0.9 },
});

try {
  // Send the command to the model and wait for the response
  const response = await client.send(command);

  // Extract and print the response text.
  const responseText = response.output.message.content[0].text;
  console.log(responseText);
} catch (err) {
  console.log(`ERROR: Can't invoke '${modelId}'. Reason: ${err}`);
  process.exit(1);
}
```

- For API details, see [Converse](#) in *AWS SDK for JavaScript API Reference*.

InvokeModel

The following code example shows how to send a text message to AI21 Labs Jurassic-2, using the Invoke Model API.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Use the Invoke Model API to send a text message.


```
import { fileURLToPath } from "url";

import { FoundationModels } from "../../config/foundation_models.js";
import {
  BedrockRuntimeClient,
  InvokeModelCommand,
} from "@aws-sdk/client-bedrock-runtime";

/**
 * @typedef {Object} Data
 * @property {string} text
 *
 * @typedef {Object} Completion
 * @property {Data} data
 *
 * @typedef {Object} ResponseBody
 * @property {Completion[]} completions
 */

/**
 * Invokes an AI21 Labs Jurassic-2 model.
 *
 * @param {string} prompt - The input text prompt for the model to complete.
 * @param {string} [modelId] - The ID of the model to use. Defaults to "ai21.j2-mid-v1".
 */
export const invokeModel = async (prompt, modelId = "ai21.j2-mid-v1") => {
  // Create a new Bedrock Runtime client instance.
  const client = new BedrockRuntimeClient({ region: "us-east-1" });

  // Prepare the payload for the model.
  const payload = {
    prompt,
    maxTokens: 500,
    temperature: 0.5,
  };

  // Invoke the model with the payload and wait for the response.
  const command = new InvokeModelCommand({
    contentType: "application/json",
    body: JSON.stringify(payload),
    modelId,
  });
}
```

```
});
const apiResponse = await client.send(command);

// Decode and return the response(s).
const decodedResponseBody = new TextDecoder().decode(apiResponse.body);
/** @type {ResponseBody} */
const responseBody = JSON.parse(decodedResponseBody);
return responseBody.completions[0].data.text;
};

// Invoke the function if this file was run directly.
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  const prompt =
    'Complete the following in one sentence: "Once upon a time...";
  const modelId = FoundationModels.JURASSIC2_MID.modelId;
  console.log(`Prompt: ${prompt}`);
  console.log(`Model ID: ${modelId}`);

  try {
    console.log("-".repeat(53));
    const response = await invokeModel(prompt, modelId);
    console.log(response);
  } catch (err) {
    console.log(err);
  }
}
```

- For API details, see [InvokeModel](#) in *AWS SDK for JavaScript API Reference*.

Amazon Titan Text

Converse

The following code example shows how to send a text message to Amazon Titan Text, using Bedrock's Converse API.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Send a text message to Amazon Titan Text, using Bedrock's Converse API.

```
// Use the Conversation API to send a text message to Amazon Titan Text.

import {
  BedrockRuntimeClient,
  ConverseCommand,
} from "@aws-sdk/client-bedrock-runtime";

// Create a Bedrock Runtime client in the AWS Region you want to use.
const client = new BedrockRuntimeClient({ region: "us-east-1" });

// Set the model ID, e.g., Titan Text Premier.
const modelId = "amazon.titan-text-premier-v1:0";

// Start a conversation with the user message.
const userMessage =
  "Describe the purpose of a 'hello world' program in one line.";
const conversation = [
  {
    role: "user",
    content: [{ text: userMessage }],
  },
];

// Create a command with the model ID, the message, and a basic configuration.
const command = new ConverseCommand({
  modelId,
  messages: conversation,
  inferenceConfig: { maxTokens: 512, temperature: 0.5, topP: 0.9 },
});

try {
  // Send the command to the model and wait for the response
  const response = await client.send(command);
}
```

```
// Extract and print the response text.
const responseText = response.output.message.content[0].text;
console.log(responseText);
} catch (err) {
  console.log(`ERROR: Can't invoke '${modelId}'. Reason: ${err}`);
  process.exit(1);
}
```

- For API details, see [Converse](#) in *AWS SDK for JavaScript API Reference*.

ConverseStream

The following code example shows how to send a text message to Amazon Titan Text, using Bedrock's Converse API and process the response stream in real-time.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Send a text message to Amazon Titan Text, using Bedrock's Converse API and process the response stream in real-time.

```
// Use the Conversation API to send a text message to Amazon Titan Text.

import {
  BedrockRuntimeClient,
  ConverseStreamCommand,
} from "@aws-sdk/client-bedrock-runtime";

// Create a Bedrock Runtime client in the AWS Region you want to use.
const client = new BedrockRuntimeClient({ region: "us-east-1" });

// Set the model ID, e.g., Titan Text Premier.
const modelId = "amazon.titan-text-premier-v1:0";
```

```
// Start a conversation with the user message.
const userMessage =
  "Describe the purpose of a 'hello world' program in one line.";
const conversation = [
  {
    role: "user",
    content: [{ text: userMessage }],
  },
];

// Create a command with the model ID, the message, and a basic configuration.
const command = new ConverseStreamCommand({
  modelId,
  messages: conversation,
  inferenceConfig: { maxTokens: 512, temperature: 0.5, topP: 0.9 },
});

try {
  // Send the command to the model and wait for the response
  const response = await client.send(command);

  // Extract and print the streamed response text in real-time.
  for await (const item of response.stream) {
    if (item.contentBlockDelta) {
      process.stdout.write(item.contentBlockDelta.delta?.text);
    }
  }
} catch (err) {
  console.log(`ERROR: Can't invoke '${modelId}'. Reason: ${err}`);
  process.exit(1);
}
```

- For API details, see [ConverseStream](#) in *AWS SDK for JavaScript API Reference*.

InvokeModel

The following code example shows how to send a text message to Amazon Titan Text, using the Invoke Model API.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Use the Invoke Model API to send a text message.

```
import { fileURLToPath } from "url";

import { FoundationModels } from "../../config/foundation_models.js";
import {
  BedrockRuntimeClient,
  InvokeModelCommand,
} from "@aws-sdk/client-bedrock-runtime";

/**
 * @typedef {Object} ResponseBody
 * @property {Object[]} results
 */

/**
 * Invokes an Amazon Titan Text generation model.
 *
 * @param {string} prompt - The input text prompt for the model to complete.
 * @param {string} [modelId] - The ID of the model to use. Defaults to
 "amazon.titan-text-express-v1".
 */
export const invokeModel = async (
  prompt,
  modelId = "amazon.titan-text-express-v1",
) => {
  // Create a new Bedrock Runtime client instance.
  const client = new BedrockRuntimeClient({ region: "us-east-1" });

  // Prepare the payload for the model.
  const payload = {
    inputText: prompt,
    textGenerationConfig: {
      maxTokenCount: 4096,
    },
  };
};
```

```
        stopSequences: [],
        temperature: 0,
        topP: 1,
    },
};

// Invoke the model with the payload and wait for the response.
const command = new InvokeModelCommand({
    contentType: "application/json",
    body: JSON.stringify(payload),
    modelId,
});
const apiResponse = await client.send(command);

// Decode and return the response.
const decodedResponseBody = new TextDecoder().decode(apiResponse.body);
/** @type {ResponseBody} */
const responseBody = JSON.parse(decodedResponseBody);
return responseBody.results[0].outputText;
};

// Invoke the function if this file was run directly.
if (process.argv[1] === fileURLToPath(import.meta.url)) {
    const prompt =
        'Complete the following in one sentence: "Once upon a time...";
    const modelId = FoundationModels.TITAN_TEXT_G1_EXPRESS.modelId;
    console.log(`Prompt: ${prompt}`);
    console.log(`Model ID: ${modelId}`);

    try {
        console.log("-".repeat(53));
        const response = await invokeModel(prompt, modelId);
        console.log(response);
    } catch (err) {
        console.log(err);
    }
}
```

- For API details, see [InvokeModel](#) in *AWS SDK for JavaScript API Reference*.

Anthropic Claude

Converse

The following code example shows how to send a text message to Anthropic Claude, using Bedrock's Converse API.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Send a text message to Anthropic Claude, using Bedrock's Converse API.

```
// Use the Conversation API to send a text message to Anthropic Claude.

import {
  BedrockRuntimeClient,
  ConverseCommand,
} from "@aws-sdk/client-bedrock-runtime";

// Create a Bedrock Runtime client in the AWS Region you want to use.
const client = new BedrockRuntimeClient({ region: "us-east-1" });

// Set the model ID, e.g., Claude 3 Haiku.
const modelId = "anthropic.claude-3-haiku-20240307-v1:0";

// Start a conversation with the user message.
const userMessage =
  "Describe the purpose of a 'hello world' program in one line.";
const conversation = [
  {
    role: "user",
    content: [{ text: userMessage }],
  },
];

// Create a command with the model ID, the message, and a basic configuration.
const command = new ConverseCommand({
```



```
    modelId,  
    messages: conversation,  
    inferenceConfig: { maxTokens: 512, temperature: 0.5, topP: 0.9 },  
  });  
  
  try {  
    // Send the command to the model and wait for the response  
    const response = await client.send(command);  
  
    // Extract and print the response text.  
    const responseText = response.output.message.content[0].text;  
    console.log(responseText);  
  } catch (err) {  
    console.log(`ERROR: Can't invoke '${modelId}'. Reason: ${err}`);  
    process.exit(1);  
  }  
}
```

- For API details, see [Converse](#) in *AWS SDK for JavaScript API Reference*.

ConverseStream

The following code example shows how to send a text message to Anthropic Claude, using Bedrock's Converse API and process the response stream in real-time.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Send a text message to Anthropic Claude, using Bedrock's Converse API and process the response stream in real-time.

```
// Use the Conversation API to send a text message to Anthropic Claude.  
  
import {  
  BedrockRuntimeClient,
```

```
    ConverseStreamCommand,
  } from "@aws-sdk/client-bedrock-runtime";

// Create a Bedrock Runtime client in the AWS Region you want to use.
const client = new BedrockRuntimeClient({ region: "us-east-1" });

// Set the model ID, e.g., Claude 3 Haiku.
const modelId = "anthropic.claude-3-haiku-20240307-v1:0";

// Start a conversation with the user message.
const userMessage =
  "Describe the purpose of a 'hello world' program in one line.";
const conversation = [
  {
    role: "user",
    content: [{ text: userMessage }],
  },
];

// Create a command with the model ID, the message, and a basic configuration.
const command = new ConverseStreamCommand({
  modelId,
  messages: conversation,
  inferenceConfig: { maxTokens: 512, temperature: 0.5, topP: 0.9 },
});

try {
  // Send the command to the model and wait for the response
  const response = await client.send(command);

  // Extract and print the streamed response text in real-time.
  for await (const item of response.stream) {
    if (item.contentBlockDelta) {
      process.stdout.write(item.contentBlockDelta.delta?.text);
    }
  }
} catch (err) {
  console.log(`ERROR: Can't invoke '${modelId}'. Reason: ${err}`);
  process.exit(1);
}
```

- For API details, see [ConverseStream](#) in *AWS SDK for JavaScript API Reference*.

InvokeModel

The following code example shows how to send a text message to Anthropic Claude, using the Invoke Model API.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Use the Invoke Model API to send a text message.

```
import { fileURLToPath } from "url";

import { FoundationModels } from "../../config/foundation_models.js";
import {
  BedrockRuntimeClient,
  InvokeModelCommand,
  InvokeModelWithResponseStreamCommand,
} from "@aws-sdk/client-bedrock-runtime";

/**
 * @typedef {Object} ResponseContent
 * @property {string} text
 *
 * @typedef {Object} MessagesResponseBody
 * @property {ResponseContent[]} content
 *
 * @typedef {Object} Delta
 * @property {string} text
 *
 * @typedef {Object} Message
 * @property {string} role
 *
 * @typedef {Object} Chunk
 * @property {string} type
 * @property {Delta} delta
 * @property {Message} message
 */
```

```
/**
 * Invokes Anthropic Claude 3 using the Messages API.
 *
 * To learn more about the Anthropic Messages API, go to:
 * https://docs.aws.amazon.com/bedrock/latest/userguide/model-parameters-anthropic-
claude-messages.html
 *
 * @param {string} prompt - The input text prompt for the model to complete.
 * @param {string} [modelId] - The ID of the model to use. Defaults to
"anthropic.claude-3-haiku-20240307-v1:0".
 */
export const invokeModel = async (
  prompt,
  modelId = "anthropic.claude-3-haiku-20240307-v1:0",
) => {
  // Create a new Bedrock Runtime client instance.
  const client = new BedrockRuntimeClient({ region: "us-east-1" });

  // Prepare the payload for the model.
  const payload = {
    anthropic_version: "bedrock-2023-05-31",
    max_tokens: 1000,
    messages: [
      {
        role: "user",
        content: [{ type: "text", text: prompt }],
      },
    ],
  };

  // Invoke Claude with the payload and wait for the response.
  const command = new InvokeModelCommand({
    contentType: "application/json",
    body: JSON.stringify(payload),
    modelId,
  });
  const apiResponse = await client.send(command);

  // Decode and return the response(s)
  const decodedResponseBody = new TextDecoder().decode(apiResponse.body);
  /** @type {MessagesResponseBody} */
  const responseBody = JSON.parse(decodedResponseBody);
  return responseBody.content[0].text;
}
```

```
};

/**
 * Invokes Anthropic Claude 3 and processes the response stream.
 *
 * To learn more about the Anthropic Messages API, go to:
 * https://docs.aws.amazon.com/bedrock/latest/userguide/model-parameters-anthropic-
claude-messages.html
 *
 * @param {string} prompt - The input text prompt for the model to complete.
 * @param {string} [modelId] - The ID of the model to use. Defaults to
"anthropic.claude-3-haiku-20240307-v1:0".
 */
export const invokeModelWithResponseStream = async (
  prompt,
  modelId = "anthropic.claude-3-haiku-20240307-v1:0",
) => {
  // Create a new Bedrock Runtime client instance.
  const client = new BedrockRuntimeClient({ region: "us-east-1" });

  // Prepare the payload for the model.
  const payload = {
    anthropic_version: "bedrock-2023-05-31",
    max_tokens: 1000,
    messages: [
      {
        role: "user",
        content: [{ type: "text", text: prompt }],
      },
    ],
  };

  // Invoke Claude with the payload and wait for the API to respond.
  const command = new InvokeModelWithResponseStreamCommand({
    contentType: "application/json",
    body: JSON.stringify(payload),
    modelId,
  });
  const apiResponse = await client.send(command);

  let completeMessage = "";

  // Decode and process the response stream
  for await (const item of apiResponse.body) {
```

```
/** @type Chunk */
const chunk = JSON.parse(new TextDecoder().decode(item.chunk.bytes));
const chunk_type = chunk.type;

if (chunk_type === "content_block_delta") {
  const text = chunk.delta.text;
  completeMessage = completeMessage + text;
  process.stdout.write(text);
}
}

// Return the final response
return completeMessage;
};

// Invoke the function if this file was run directly.
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  const prompt = 'Write a paragraph starting with: "Once upon a time...";
  const modelId = FoundationModels.CLAUDE_3_HAIKU.modelId;
  console.log(`Prompt: ${prompt}`);
  console.log(`Model ID: ${modelId}`);

  try {
    console.log("-".repeat(53));
    const response = await invokeModel(prompt, modelId);
    console.log("\n" + "-".repeat(53));
    console.log("Final structured response:");
    console.log(response);
  } catch (err) {
    console.log(`\n${err}`);
  }
}
```

- For API details, see [InvokeModel](#) in *AWS SDK for JavaScript API Reference*.

InvokeModelWithResponseStream

The following code example shows how to send a text message to Anthropic Claude models, using the Invoke Model API, and print the response stream.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Use the Invoke Model API to send a text message and process the response stream in real-time.

```
import { fileURLToPath } from "url";

import { FoundationModels } from "../../config/foundation_models.js";
import {
  BedrockRuntimeClient,
  InvokeModelCommand,
  InvokeModelWithResponseStreamCommand,
} from "@aws-sdk/client-bedrock-runtime";

/**
 * @typedef {Object} ResponseContent
 * @property {string} text
 *
 * @typedef {Object} MessagesResponseBody
 * @property {ResponseContent[]} content
 *
 * @typedef {Object} Delta
 * @property {string} text
 *
 * @typedef {Object} Message
 * @property {string} role
 *
 * @typedef {Object} Chunk
 * @property {string} type
 * @property {Delta} delta
 * @property {Message} message
 */

/**
 * Invokes Anthropic Claude 3 using the Messages API.
 *
 * To learn more about the Anthropic Messages API, go to:
```

```
* https://docs.aws.amazon.com/bedrock/latest/userguide/model-parameters-anthropic-claude-messages.html
*
* @param {string} prompt - The input text prompt for the model to complete.
* @param {string} [modelId] - The ID of the model to use. Defaults to
"anthropic.claude-3-haiku-20240307-v1:0".
*/
export const invokeModel = async (
  prompt,
  modelId = "anthropic.claude-3-haiku-20240307-v1:0",
) => {
  // Create a new Bedrock Runtime client instance.
  const client = new BedrockRuntimeClient({ region: "us-east-1" });

  // Prepare the payload for the model.
  const payload = {
    anthropic_version: "bedrock-2023-05-31",
    max_tokens: 1000,
    messages: [
      {
        role: "user",
        content: [{ type: "text", text: prompt }],
      },
    ],
  };

  // Invoke Claude with the payload and wait for the response.
  const command = new InvokeModelCommand({
    contentType: "application/json",
    body: JSON.stringify(payload),
    modelId,
  });
  const apiResponse = await client.send(command);

  // Decode and return the response(s)
  const decodedResponseBody = new TextDecoder().decode(apiResponse.body);
  /** @type {MessagesResponseBody} */
  const responseBody = JSON.parse(decodedResponseBody);
  return responseBody.content[0].text;
};

/**
 * Invokes Anthropic Claude 3 and processes the response stream.
 */
```



```
* To learn more about the Anthropic Messages API, go to:
* https://docs.aws.amazon.com/bedrock/latest/userguide/model-parameters-anthropic-claude-messages.html
*
* @param {string} prompt - The input text prompt for the model to complete.
* @param {string} [modelId] - The ID of the model to use. Defaults to
"anthropic.claude-3-haiku-20240307-v1:0".
*/
export const invokeModelWithResponseStream = async (
  prompt,
  modelId = "anthropic.claude-3-haiku-20240307-v1:0",
) => {
  // Create a new Bedrock Runtime client instance.
  const client = new BedrockRuntimeClient({ region: "us-east-1" });

  // Prepare the payload for the model.
  const payload = {
    anthropic_version: "bedrock-2023-05-31",
    max_tokens: 1000,
    messages: [
      {
        role: "user",
        content: [{ type: "text", text: prompt }],
      },
    ],
  };

  // Invoke Claude with the payload and wait for the API to respond.
  const command = new InvokeModelWithResponseStreamCommand({
    contentType: "application/json",
    body: JSON.stringify(payload),
    modelId,
  });
  const apiResponse = await client.send(command);

  let completeMessage = "";

  // Decode and process the response stream
  for await (const item of apiResponse.body) {
    /** @type Chunk */
    const chunk = JSON.parse(new TextDecoder().decode(item.chunk.bytes));
    const chunk_type = chunk.type;

    if (chunk_type === "content_block_delta") {
```

```
        const text = chunk.delta.text;
        completeMessage = completeMessage + text;
        process.stdout.write(text);
    }
}

// Return the final response
return completeMessage;
};

// Invoke the function if this file was run directly.
if (process.argv[1] === fileURLToPath(import.meta.url)) {
    const prompt = 'Write a paragraph starting with: "Once upon a time...";
    const modelId = FoundationModels.CLAUDE_3_HAIKU.modelId;
    console.log(`Prompt: ${prompt}`);
    console.log(`Model ID: ${modelId}`);

    try {
        console.log("-".repeat(53));
        const response = await invokeModel(prompt, modelId);
        console.log("\n" + "-".repeat(53));
        console.log("Final structured response:");
        console.log(response);
    } catch (err) {
        console.log(`\n${err}`);
    }
}
```

- For API details, see [InvokeModelWithResponseStream](#) in *AWS SDK for JavaScript API Reference*.

Cohere Command

Converse

The following code example shows how to send a text message to Cohere Command, using Bedrock's Converse API.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Send a text message to Cohere Command, using Bedrock's Converse API.

```
// Use the Conversation API to send a text message to Cohere Command.

import {
  BedrockRuntimeClient,
  ConverseCommand,
} from "@aws-sdk/client-bedrock-runtime";

// Create a Bedrock Runtime client in the AWS Region you want to use.
const client = new BedrockRuntimeClient({ region: "us-east-1" });

// Set the model ID, e.g., Command R.
const modelId = "cohere.command-r-v1:0";

// Start a conversation with the user message.
const userMessage =
  "Describe the purpose of a 'hello world' program in one line.";
const conversation = [
  {
    role: "user",
    content: [{ text: userMessage }],
  },
];

// Create a command with the model ID, the message, and a basic configuration.
const command = new ConverseCommand({
  modelId,
  messages: conversation,
  inferenceConfig: { maxTokens: 512, temperature: 0.5, topP: 0.9 },
});

try {
  // Send the command to the model and wait for the response
  const response = await client.send(command);
}
```

```
// Extract and print the response text.
const responseText = response.output.message.content[0].text;
console.log(responseText);
} catch (err) {
  console.log(`ERROR: Can't invoke '${modelId}'. Reason: ${err}`);
  process.exit(1);
}
```

- For API details, see [Converse](#) in *AWS SDK for JavaScript API Reference*.

ConverseStream

The following code example shows how to send a text message to Cohere Command, using Bedrock's Converse API and process the response stream in real-time.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Send a text message to Cohere Command, using Bedrock's Converse API and process the response stream in real-time.

```
// Use the Conversation API to send a text message to Cohere Command.

import {
  BedrockRuntimeClient,
  ConverseStreamCommand,
} from "@aws-sdk/client-bedrock-runtime";

// Create a Bedrock Runtime client in the AWS Region you want to use.
const client = new BedrockRuntimeClient({ region: "us-east-1" });

// Set the model ID, e.g., Command R.
const modelId = "cohere.command-r-v1:0";
```

```
// Start a conversation with the user message.
const userMessage =
  "Describe the purpose of a 'hello world' program in one line.";
const conversation = [
  {
    role: "user",
    content: [{ text: userMessage }],
  },
];

// Create a command with the model ID, the message, and a basic configuration.
const command = new ConverseStreamCommand({
  modelId,
  messages: conversation,
  inferenceConfig: { maxTokens: 512, temperature: 0.5, topP: 0.9 },
});

try {
  // Send the command to the model and wait for the response
  const response = await client.send(command);

  // Extract and print the streamed response text in real-time.
  for await (const item of response.stream) {
    if (item.contentBlockDelta) {
      process.stdout.write(item.contentBlockDelta.delta?.text);
    }
  }
} catch (err) {
  console.log(`ERROR: Can't invoke '${modelId}'. Reason: ${err}`);
  process.exit(1);
}
```

- For API details, see [ConverseStream](#) in *AWS SDK for JavaScript API Reference*.

Meta Llama

Converse

The following code example shows how to send a text message to Meta Llama, using Bedrock's Converse API.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Send a text message to Meta Llama, using Bedrock's Converse API.

```
// Use the Conversation API to send a text message to Meta Llama.

import {
  BedrockRuntimeClient,
  ConverseCommand,
} from "@aws-sdk/client-bedrock-runtime";

// Create a Bedrock Runtime client in the AWS Region you want to use.
const client = new BedrockRuntimeClient({ region: "us-east-1" });

// Set the model ID, e.g., Llama 3 8b Instruct.
const modelId = "meta.llama3-8b-instruct-v1:0";

// Start a conversation with the user message.
const userMessage =
  "Describe the purpose of a 'hello world' program in one line.";
const conversation = [
  {
    role: "user",
    content: [{ text: userMessage }],
  },
];

// Create a command with the model ID, the message, and a basic configuration.
const command = new ConverseCommand({
  modelId,
  messages: conversation,
  inferenceConfig: { maxTokens: 512, temperature: 0.5, topP: 0.9 },
});

try {
  // Send the command to the model and wait for the response
  const response = await client.send(command);
}
```

```
// Extract and print the response text.
const responseText = response.output.message.content[0].text;
console.log(responseText);
} catch (err) {
  console.log(`ERROR: Can't invoke '${modelId}'. Reason: ${err}`);
  process.exit(1);
}
```

- For API details, see [Converse](#) in *AWS SDK for JavaScript API Reference*.

ConverseStream

The following code example shows how to send a text message to Meta Llama, using Bedrock's Converse API and process the response stream in real-time.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Send a text message to Meta Llama, using Bedrock's Converse API and process the response stream in real-time.

```
// Use the Conversation API to send a text message to Meta Llama.

import {
  BedrockRuntimeClient,
  ConverseStreamCommand,
} from "@aws-sdk/client-bedrock-runtime";

// Create a Bedrock Runtime client in the AWS Region you want to use.
const client = new BedrockRuntimeClient({ region: "us-east-1" });

// Set the model ID, e.g., Llama 3 8b Instruct.
const modelId = "meta.llama3-8b-instruct-v1:0";
```

```
// Start a conversation with the user message.
const userMessage =
  "Describe the purpose of a 'hello world' program in one line.";
const conversation = [
  {
    role: "user",
    content: [{ text: userMessage }],
  },
];

// Create a command with the model ID, the message, and a basic configuration.
const command = new ConverseStreamCommand({
  modelId,
  messages: conversation,
  inferenceConfig: { maxTokens: 512, temperature: 0.5, topP: 0.9 },
});

try {
  // Send the command to the model and wait for the response
  const response = await client.send(command);

  // Extract and print the streamed response text in real-time.
  for await (const item of response.stream) {
    if (item.contentBlockDelta) {
      process.stdout.write(item.contentBlockDelta.delta?.text);
    }
  }
} catch (err) {
  console.log(`ERROR: Can't invoke '${modelId}'. Reason: ${err}`);
  process.exit(1);
}
```

- For API details, see [ConverseStream](#) in *AWS SDK for JavaScript API Reference*.

InvokeModel: Llama 2

The following code example shows how to send a text message to Meta Llama 2, using the Invoke Model API.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Use the Invoke Model API to send a text message.

```
// Send a prompt to Meta Llama 2 and print the response.

import {
  BedrockRuntimeClient,
  InvokeModelCommand,
} from "@aws-sdk/client-bedrock-runtime";

// Create a Bedrock Runtime client in the AWS Region of your choice.
const client = new BedrockRuntimeClient({ region: "us-west-2" });

// Set the model ID, e.g., Llama 2 Chat 13B.
const modelId = "meta.llama2-13b-chat-v1";

// Define the user message to send.
const userMessage =
  "Describe the purpose of a 'hello world' program in one sentence.";

// Embed the message in Llama 2's prompt format.
const prompt = `[INST] ${userMessage} [/INST]`;

// Format the request payload using the model's native structure.
const request = {
  prompt,
  // Optional inference parameters:
  max_gen_len: 512,
  temperature: 0.5,
  top_p: 0.9,
};

// Encode and send the request.
const response = await client.send(
  new InvokeModelCommand({
    contentType: "application/json",
```

```
    body: JSON.stringify(request,
      modelId,
    )),
  );

// Decode the native response body.
/** @type {{ generation: string }} */
const nativeResponse = JSON.parse(new TextDecoder().decode(response.body));

// Extract and print the generated text.
const responseText = nativeResponse.generation;
console.log(responseText);

// Learn more about the Llama 2 prompt format at:
// https://llama.meta.com/docs/model-cards-and-prompt-formats/meta-llama-2
```

- For API details, see [InvokeModel](#) in *AWS SDK for JavaScript API Reference*.

InvokeModel: Llama 3

The following code example shows how to send a text message to Meta Llama 3, using the Invoke Model API.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Use the Invoke Model API to send a text message.

```
// Send a prompt to Meta Llama 3 and print the response.

import {
  BedrockRuntimeClient,
  InvokeModelCommand,
} from "@aws-sdk/client-bedrock-runtime";
```

```
// Create a Bedrock Runtime client in the AWS Region of your choice.
const client = new BedrockRuntimeClient({ region: "us-west-2" });

// Set the model ID, e.g., Llama 3 8B Instruct.
const modelId = "meta.llama3-8b-instruct-v1:0";

// Define the user message to send.
const userMessage =
  "Describe the purpose of a 'hello world' program in one sentence.";

// Embed the message in Llama 3's prompt format.
const prompt = `
<|begin_of_text|>
<|start_header_id|>user<|end_header_id|>
${userMessage}
<|eot_id|>
<|start_header_id|>assistant<|end_header_id|>
`;

// Format the request payload using the model's native structure.
const request = {
  prompt,
  // Optional inference parameters:
  max_gen_len: 512,
  temperature: 0.5,
  top_p: 0.9,
};

// Encode and send the request.
const response = await client.send(
  new InvokeModelCommand({
    contentType: "application/json",
    body: JSON.stringify(request),
    modelId,
  }),
);

// Decode the native response body.
/** @type {{ generation: string }} */
const nativeResponse = JSON.parse(new TextDecoder().decode(response.body));

// Extract and print the generated text.
const responseText = nativeResponse.generation;
console.log(responseText);
```

```
// Learn more about the Llama 3 prompt format at:  
// https://llama.meta.com/docs/model-cards-and-prompt-formats/meta-llama-3/#special-tokens-used-with-meta-llama-3
```

- For API details, see [InvokeModel](#) in *AWS SDK for JavaScript API Reference*.

InvokeModelWithResponseStream: Llama 2

The following code example shows how to send a text message to Meta Llama 2, using the Invoke Model API, and print the response stream.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Use the Invoke Model API to send a text message and process the response stream in real-time.

```
// Send a prompt to Meta Llama 2 and print the response stream in real-time.  
  
import {  
  BedrockRuntimeClient,  
  InvokeModelWithResponseStreamCommand,  
} from "@aws-sdk/client-bedrock-runtime";  
  
// Create a Bedrock Runtime client in the AWS Region of your choice.  
const client = new BedrockRuntimeClient({ region: "us-west-2" });  
  
// Set the model ID, e.g., Llama 2 Chat 13B.  
const modelId = "meta.llama2-13b-chat-v1";  
  
// Define the user message to send.  
const userMessage =  
  "Describe the purpose of a 'hello world' program in one sentence.";  
  
// Embed the message in Llama 2's prompt format.
```

```
const prompt = `[INST] ${userMessage} [/INST]`;

// Format the request payload using the model's native structure.
const request = {
  prompt,
  // Optional inference parameters:
  max_gen_len: 512,
  temperature: 0.5,
  top_p: 0.9,
};

// Encode and send the request.
const responseStream = await client.send(
  new InvokeModelWithResponseStreamCommand({
    contentType: "application/json",
    body: JSON.stringify(request),
    modelId,
  }),
);

// Extract and print the response stream in real-time.
for await (const event of responseStream.body) {
  /** @type {{ generation: string }} */
  const chunk = JSON.parse(new TextDecoder().decode(event.chunk.bytes));
  if (chunk.generation) {
    process.stdout.write(chunk.generation);
  }
}

// Learn more about the Llama 3 prompt format at:
// https://llama.meta.com/docs/model-cards-and-prompt-formats/meta-llama-3/#special-tokens-used-with-meta-llama-3
```

- For API details, see [InvokeModelWithResponseStream](#) in *AWS SDK for JavaScript API Reference*.

InvokeModelWithResponseStream: Llama 3

The following code example shows how to send a text message to Meta Llama 3, using the Invoke Model API, and print the response stream.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Use the Invoke Model API to send a text message and process the response stream in real-time.

```
// Send a prompt to Meta Llama 3 and print the response stream in real-time.

import {
  BedrockRuntimeClient,
  InvokeModelWithResponseStreamCommand,
} from "@aws-sdk/client-bedrock-runtime";

// Create a Bedrock Runtime client in the AWS Region of your choice.
const client = new BedrockRuntimeClient({ region: "us-west-2" });

// Set the model ID, e.g., Llama 3 8B Instruct.
const modelId = "meta.llama3-8b-instruct-v1:0";

// Define the user message to send.
const userMessage =
  "Describe the purpose of a 'hello world' program in one sentence.";

// Embed the message in Llama 3's prompt format.
const prompt = `
<|begin_of_text|>
<|start_header_id|>user<|end_header_id|>
${userMessage}
<|eot_id|>
<|start_header_id|>assistant<|end_header_id|>
`;

// Format the request payload using the model's native structure.
const request = {
  prompt,
  // Optional inference parameters:
  max_gen_len: 512,
  temperature: 0.5,
  top_p: 0.9,
```

```
};

// Encode and send the request.
const responseStream = await client.send(
  new InvokeModelWithResponseStreamCommand({
    contentType: "application/json",
    body: JSON.stringify(request),
    modelId,
  }),
);

// Extract and print the response stream in real-time.
for await (const event of responseStream.body) {
  /** @type {{ generation: string }} */
  const chunk = JSON.parse(new TextDecoder().decode(event.chunk.bytes));
  if (chunk.generation) {
    process.stdout.write(chunk.generation);
  }
}

// Learn more about the Llama 3 prompt format at:
// https://llama.meta.com/docs/model-cards-and-prompt-formats/meta-llama-3/#special-tokens-used-with-meta-llama-3
```

- For API details, see [InvokeModelWithResponseStream](#) in *AWS SDK for JavaScript API Reference*.

Mistral AI

Converse

The following code example shows how to send a text message to Mistral, using Bedrock's Converse API.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Send a text message to Mistral, using Bedrock's Converse API.

```
// Use the Conversation API to send a text message to Mistral.

import {
  BedrockRuntimeClient,
  ConverseCommand,
} from "@aws-sdk/client-bedrock-runtime";

// Create a Bedrock Runtime client in the AWS Region you want to use.
const client = new BedrockRuntimeClient({ region: "us-east-1" });

// Set the model ID, e.g., Mistral Large.
const modelId = "mistral.mistral-large-2402-v1:0";

// Start a conversation with the user message.
const userMessage =
  "Describe the purpose of a 'hello world' program in one line.";
const conversation = [
  {
    role: "user",
    content: [{ text: userMessage }],
  },
];

// Create a command with the model ID, the message, and a basic configuration.
const command = new ConverseCommand({
  modelId,
  messages: conversation,
  inferenceConfig: { maxTokens: 512, temperature: 0.5, topP: 0.9 },
});

try {
  // Send the command to the model and wait for the response
  const response = await client.send(command);

  // Extract and print the response text.
  const responseText = response.output.message.content[0].text;
  console.log(responseText);
} catch (err) {
  console.log(`ERROR: Can't invoke '${modelId}'. Reason: ${err}`);
  process.exit(1);
}
```


- For API details, see [Converse](#) in *AWS SDK for JavaScript API Reference*.

ConverseStream

The following code example shows how to send a text message to Mistral, using Bedrock's Converse API and process the response stream in real-time.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Send a text message to Mistral, using Bedrock's Converse API and process the response stream in real-time.

```
// Use the Conversation API to send a text message to Mistral.

import {
  BedrockRuntimeClient,
  ConverseStreamCommand,
} from "@aws-sdk/client-bedrock-runtime";

// Create a Bedrock Runtime client in the AWS Region you want to use.
const client = new BedrockRuntimeClient({ region: "us-east-1" });

// Set the model ID, e.g., Mistral Large.
const modelId = "mistral.mistral-large-2402-v1:0";

// Start a conversation with the user message.
const userMessage =
  "Describe the purpose of a 'hello world' program in one line.";
const conversation = [
  {
    role: "user",
    content: [{ text: userMessage }],
  },
];
```

```
// Create a command with the model ID, the message, and a basic configuration.
const command = new ConverseStreamCommand({
  modelId,
  messages: conversation,
  inferenceConfig: { maxTokens: 512, temperature: 0.5, topP: 0.9 },
});

try {
  // Send the command to the model and wait for the response
  const response = await client.send(command);

  // Extract and print the streamed response text in real-time.
  for await (const item of response.stream) {
    if (item.contentBlockDelta) {
      process.stdout.write(item.contentBlockDelta.delta?.text);
    }
  }
} catch (err) {
  console.log(`ERROR: Can't invoke '${modelId}'. Reason: ${err}`);
  process.exit(1);
}
```

- For API details, see [ConverseStream](#) in *AWS SDK for JavaScript API Reference*.

InvokeModel

The following code example shows how to send a text message to Mistral models, using the Invoke Model API.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Use the Invoke Model API to send a text message.

```
import { fileURLToPath } from "url";

import { FoundationModels } from "../../config/foundation_models.js";
import {
  BedrockRuntimeClient,
  InvokeModelCommand,
} from "@aws-sdk/client-bedrock-runtime";

/**
 * @typedef {Object} Output
 * @property {string} text
 *
 * @typedef {Object} ResponseBody
 * @property {Output[]} outputs
 */

/**
 * Invokes a Mistral 7B Instruct model.
 *
 * @param {string} prompt - The input text prompt for the model to complete.
 * @param {string} [modelId] - The ID of the model to use. Defaults to
 * "mistral.mistral-7b-instruct-v0:2".
 */
export const invokeModel = async (
  prompt,
  modelId = "mistral.mistral-7b-instruct-v0:2",
) => {
  // Create a new Bedrock Runtime client instance.
  const client = new BedrockRuntimeClient({ region: "us-east-1" });

  // Mistral instruct models provide optimal results when embedding
  // the prompt into the following template:
  const instruction = `
```

```
const command = new InvokeModelCommand({
  contentType: "application/json",
  body: JSON.stringify(payload),
  modelId,
});
const apiResponse = await client.send(command);

// Decode and return the response.
const decodedResponseBody = new TextDecoder().decode(apiResponse.body);
/** @type {ResponseBody} */
const responseBody = JSON.parse(decodedResponseBody);
return responseBody.outputs[0].text;
};

// Invoke the function if this file was run directly.
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  const prompt =
    'Complete the following in one sentence: "Once upon a time...";
  const modelId = FoundationModels.MISTRAL_7B.modelId;
  console.log(`Prompt: ${prompt}`);
  console.log(`Model ID: ${modelId}`);

  try {
    console.log("-".repeat(53));
    const response = await invokeModel(prompt, modelId);
    console.log(response);
  } catch (err) {
    console.log(err);
  }
}
```

- For API details, see [InvokeModel](#) in *AWS SDK for JavaScript API Reference*.

Agents for Amazon Bedrock examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with Agents for Amazon Bedrock.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Get started

Hello Agents for Amazon Bedrock

The following code example shows how to get started using Agents for Amazon Bedrock.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { fileURLToPath } from "url";

import {
  BedrockAgentClient,
  GetAgentCommand,
  paginateListAgents,
} from "@aws-sdk/client-bedrock-agent";

/**
 * @typedef {Object} AgentSummary
 */

/**
 * A simple scenario to demonstrate basic setup and interaction with the Bedrock
 * Agents Client.
 *
 * This function first initializes the Amazon Bedrock Agents client for a specific
 * region.
 * It then retrieves a list of existing agents using the streamlined paginator
 * approach.
 * For each agent found, it retrieves detailed information using a command object.
 */
```

```
* Demonstrates:
* - Use of the Bedrock Agents client to initialize and communicate with the AWS
service.
* - Listing resources in a paginated response pattern.
* - Accessing an individual resource using a command object.
*
* @returns {Promise<void>} A promise that resolves when the function has completed
execution.
*/
export const main = async () => {
  const region = "us-east-1";

  console.log("=".repeat(68));

  console.log(`Initializing Amazon Bedrock Agents client for ${region}...`);
  const client = new BedrockAgentClient({ region });

  console.log(`Retrieving the list of existing agents...`);
  const paginatorConfig = { client };
  const pages = paginateListAgents(paginatorConfig, {});

  /** @type {AgentSummary[]} */
  const agentSummaries = [];
  for await (const page of pages) {
    agentSummaries.push(...page.agentSummaries);
  }

  console.log(`Found ${agentSummaries.length} agents in ${region}.`);

  if (agentSummaries.length > 0) {
    for (const agentSummary of agentSummaries) {
      const agentId = agentSummary.agentId;
      console.log("=".repeat(68));
      console.log(`Retrieving agent with ID: ${agentId}:`);
      console.log("-".repeat(68));

      const command = new GetAgentCommand({ agentId });
      const response = await client.send(command);
      const agent = response.agent;

      console.log(` Name: ${agent.agentName}`);
      console.log(` Status: ${agent.agentStatus}`);
      console.log(` ARN: ${agent.agentArn}`);
      console.log(` Foundation model: ${agent.foundationModel}`);
    }
  }
}
```

```
    }  
  }  
  console.log("=".repeat(68));  
};  
  
// Invoke main function if this file was run directly.  
if (process.argv[1] === fileURLToPath(import.meta.url)) {  
  await main();  
}
```

- For API details, see the following topics in *AWS SDK for JavaScript API Reference*.
 - [GetAgent](#)
 - [ListAgents](#)

Topics

- [Actions](#)

Actions

CreateAgent

The following code example shows how to use CreateAgent.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create an agent.

```
import { fileURLToPath } from "url";  
import { checkForPlaceholders } from "../lib/utils.js";  
  
import {  
  BedrockAgentClient,
```

```
    CreateAgentCommand,
  } from "@aws-sdk/client-bedrock-agent";

/**
 * Creates an Amazon Bedrock Agent.
 *
 * @param {string} agentName - A name for the agent that you create.
 * @param {string} foundationModel - The foundation model to be used by the agent
you create.
 * @param {string} agentResourceRoleArn - The ARN of the IAM role with permissions
required by the agent.
 * @param {string} [region='us-east-1'] - The AWS region in use.
 * @returns {Promise<import("@aws-sdk/client-bedrock-agent").Agent>} An object
containing details of the created agent.
 */
export const createAgent = async (
  agentName,
  foundationModel,
  agentResourceRoleArn,
  region = "us-east-1",
) => {
  const client = new BedrockAgentClient({ region });

  const command = new CreateAgentCommand({
    agentName,
    foundationModel,
    agentResourceRoleArn,
  });
  const response = await client.send(command);

  return response.agent;
};

// Invoke main function if this file was run directly.
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  // Replace the placeholders for agentName and accountId, and roleName with a
unique name for the new agent,
  // the id of your AWS account, and the name of an existing execution role that the
agent can use inside your account.
  // For foundationModel, specify the desired model. Ensure to remove the brackets
'[]' before adding your data.

  // A string (max 100 chars) that can include letters, numbers, dashes '-', and
underscores '_'.

```



```
const agentName = "[your-bedrock-agent-name]";

// Your AWS account id.
const accountId = "[123456789012]";

// The name of the agent's execution role. It must be prefixed by
`AmazonBedrockExecutionRoleForAgents_`.
const roleName = "[AmazonBedrockExecutionRoleForAgents_your-role-name]";

// The ARN for the agent's execution role.
// Follow the ARN format: 'arn:aws:iam::account-id:role/role-name'
const roleArn = `arn:aws:iam::${accountId}:role/${roleName}`;

// Specify the model for the agent. Change if a different model is preferred.
const foundationModel = "anthropic.claude-v2";

// Check for unresolved placeholders in agentName and roleArn.
checkForPlaceholders([agentName, roleArn]);

console.log(`Creating a new agent...`);

const agent = await createAgent(agentName, foundationModel, roleArn);
console.log(agent);
}
```

- For API details, see [CreateAgent](#) in *AWS SDK for JavaScript API Reference*.

DeleteAgent

The following code example shows how to use DeleteAgent.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Delete an agent.

```
import { fileURLToPath } from "url";
import { checkForPlaceholders } from "../lib/utils.js";

import {
  BedrockAgentClient,
  DeleteAgentCommand,
} from "@aws-sdk/client-bedrock-agent";

/**
 * Deletes an Amazon Bedrock Agent.
 *
 * @param {string} agentId - The unique identifier of the agent to delete.
 * @param {string} [region='us-east-1'] - The AWS region in use.
 * @returns {Promise<import("@aws-sdk/client-bedrock-agent").DeleteAgentCommandOutput>} An object containing the agent id, the status,
  and some additional metadata.
 */
export const deleteAgent = (agentId, region = "us-east-1") => {
  const client = new BedrockAgentClient({ region });
  const command = new DeleteAgentCommand({ agentId });
  return client.send(command);
};

// Invoke main function if this file was run directly.
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  // Replace the placeholders for agentId with an existing agent's id.
  // Ensure to remove the brackets (`[]`) before adding your data.

  // The agentId must be an alphanumeric string with exactly 10 characters.
  const agentId = "[ABC123DE45]";

  // Check for unresolved placeholders in agentId.
  checkForPlaceholders([agentId]);

  console.log(`Deleting agent with ID ${agentId}...`);

  const response = await deleteAgent(agentId);
  console.log(response);
}
```

- For API details, see [DeleteAgent](#) in *AWS SDK for JavaScript API Reference*.

GetAgent

The following code example shows how to use GetAgent.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Get an agent.

```
import { fileURLToPath } from "url";
import { checkForPlaceholders } from "../lib/utils.js";

import {
  BedrockAgentClient,
  GetAgentCommand,
} from "@aws-sdk/client-bedrock-agent";

/**
 * Retrieves the details of an Amazon Bedrock Agent.
 *
 * @param {string} agentId - The unique identifier of the agent.
 * @param {string} [region='us-east-1'] - The AWS region in use.
 * @returns {Promise<import("@aws-sdk/client-bedrock-agent").Agent>} An object
  containing the agent details.
 */
export const getAgent = async (agentId, region = "us-east-1") => {
  const client = new BedrockAgentClient({ region });

  const command = new GetAgentCommand({ agentId });
  const response = await client.send(command);
  return response.agent;
};

// Invoke main function if this file was run directly.
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  // Replace the placeholders for agentId with an existing agent's id.
  // Ensure to remove the brackets '[]' before adding your data.
```

```
// The agentId must be an alphanumeric string with exactly 10 characters.
const agentId = "[ABC123DE45]";

// Check for unresolved placeholders in agentId.
checkForPlaceholders([agentId]);

console.log(`Retrieving agent with ID ${agentId}...`);

const agent = await getAgent(agentId);
console.log(agent);
}
```

- For API details, see [GetAgent](#) in *AWS SDK for JavaScript API Reference*.

ListAgentActionGroups

The following code example shows how to use `ListAgentActionGroups`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

List the action groups for an agent.

```
import { fileURLToPath } from "url";
import { checkForPlaceholders } from "../lib/utils.js";

import {
  BedrockAgentClient,
  ListAgentActionGroupsCommand,
  paginateListAgentActionGroups,
} from "@aws-sdk/client-bedrock-agent";

/**
 * Retrieves a list of Action Groups of an agent utilizing the paginator function.
```

```
*
* This function leverages a paginator, which abstracts the complexity of
* pagination, providing
* a straightforward way to handle paginated results inside a `for await...of` loop.
*
* @param {string} agentId - The unique identifier of the agent.
* @param {string} agentVersion - The version of the agent.
* @param {string} [region='us-east-1'] - The AWS region in use.
* @returns {Promise<ActionGroupSummary[]>} An array of action group summaries.
*/
export const listAgentActionGroupsWithPaginator = async (
  agentId,
  agentVersion,
  region = "us-east-1",
) => {
  const client = new BedrockAgentClient({ region });

  // Create a paginator configuration
  const paginatorConfig = {
    client,
    pageSize: 10, // optional, added for demonstration purposes
  };

  const params = { agentId, agentVersion };

  const pages = paginateListAgentActionGroups(paginatorConfig, params);

  // Paginate until there are no more results
  const actionGroupSummaries = [];
  for await (const page of pages) {
    actionGroupSummaries.push(...page.actionGroupSummaries);
  }

  return actionGroupSummaries;
};

/**
* Retrieves a list of Action Groups of an agent utilizing the
ListAgentActionGroupsCommand.
*
* This function demonstrates the manual approach, sending a command to the client
and processing the response.
* Pagination must manually be managed. For a simplified approach that abstracts
away pagination logic, see
```

```
* the `listAgentActionGroupsWithPaginator()` example below.
*
* @param {string} agentId - The unique identifier of the agent.
* @param {string} agentVersion - The version of the agent.
* @param {string} [region='us-east-1'] - The AWS region in use.
* @returns {Promise<ActionGroupSummary[]>} An array of action group summaries.
*/
export const listAgentActionGroupsWithCommandObject = async (
  agentId,
  agentVersion,
  region = "us-east-1",
) => {
  const client = new BedrockAgentClient({ region });

  let nextToken;
  const actionGroupSummaries = [];
  do {
    const command = new ListAgentActionGroupsCommand({
      agentId,
      agentVersion,
      nextToken,
      maxResults: 10, // optional, added for demonstration purposes
    });

    /** @type {{actionGroupSummaries: ActionGroupSummary[], nextToken?: string}} */
    const response = await client.send(command);

    for (const actionGroup of response.actionGroupSummaries || []) {
      actionGroupSummaries.push(actionGroup);
    }

    nextToken = response.nextToken;
  } while (nextToken);

  return actionGroupSummaries;
};

// Invoke main function if this file was run directly.
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  // Replace the placeholders for agentId and agentVersion with an existing agent's
  id and version.
  // Ensure to remove the brackets '[]' before adding your data.

  // The agentId must be an alphanumeric string with exactly 10 characters.
```

```
const agentId = "[ABC123DE45]";

// A string either containing `DRAFT` or a number with 1-5 digits (e.g., '123' or
'DRAFT').
const agentVersion = "[DRAFT]";

// Check for unresolved placeholders in agentId and agentVersion.
checkForPlaceholders([agentId, agentVersion]);

console.log("=".repeat(68));
console.log(
  "Listing agent action groups using ListAgentActionGroupsCommand:",
);

for (const actionGroup of await listAgentActionGroupsWithCommandObject(
  agentId,
  agentVersion,
)) {
  console.log(actionGroup);
}

console.log("=".repeat(68));
console.log(
  "Listing agent action groups using the paginateListAgents function:",
);
for (const actionGroup of await listAgentActionGroupsWithPaginator(
  agentId,
  agentVersion,
)) {
  console.log(actionGroup);
}
}
```

- For API details, see [ListAgentActionGroups](#) in *AWS SDK for JavaScript API Reference*.

ListAgents

The following code example shows how to use ListAgents.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

List the agents belonging to an account.

```
import { fileURLToPath } from "url";

import {
  BedrockAgentClient,
  ListAgentsCommand,
  paginateListAgents,
} from "@aws-sdk/client-bedrock-agent";

/**
 * Retrieves a list of available Amazon Bedrock agents utilizing the paginator
 * function.
 *
 * This function leverages a paginator, which abstracts the complexity of
 * pagination, providing
 * a straightforward way to handle paginated results inside a `for await...of` loop.
 *
 * @param {string} [region='us-east-1'] - The AWS region in use.
 * @returns {Promise<AgentSummary[]>} An array of agent summaries.
 */
export const listAgentsWithPaginator = async (region = "us-east-1") => {
  const client = new BedrockAgentClient({ region });

  const paginatorConfig = {
    client,
    pageSize: 10, // optional, added for demonstration purposes
  };

  const pages = paginateListAgents(paginatorConfig, {});

  // Paginate until there are no more results
  const agentSummaries = [];
  for await (const page of pages) {
```



```
    agentSummaries.push(...page.agentSummaries);
  }

  return agentSummaries;
};

/**
 * Retrieves a list of available Amazon Bedrock agents utilizing the
 * ListAgentsCommand.
 *
 * This function demonstrates the manual approach, sending a command to the client
 * and processing the response.
 *
 * Pagination must manually be managed. For a simplified approach that abstracts
 * away pagination logic, see
 * the `listAgentsWithPaginator()` example below.
 *
 * @param {string} [region='us-east-1'] - The AWS region in use.
 * @returns {Promise<AgentSummary[]>} An array of agent summaries.
 */
export const listAgentsWithCommandObject = async (region = "us-east-1") => {
  const client = new BedrockAgentClient({ region });

  let nextToken;
  const agentSummaries = [];
  do {
    const command = new ListAgentsCommand({
      nextToken,
      maxResults: 10, // optional, added for demonstration purposes
    });

    /** @type {{agentSummaries: AgentSummary[], nextToken?: string}} */
    const paginatedResponse = await client.send(command);

    agentSummaries.push(...(paginatedResponse.agentSummaries || []));

    nextToken = paginatedResponse.nextToken;
  } while (nextToken);

  return agentSummaries;
};

// Invoke main function if this file was run directly.
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  console.log("=".repeat(68));
}
```

```
console.log("Listing agents using ListAgentsCommand:");
for (const agent of await listAgentsWithCommandObject()) {
  console.log(agent);
}

console.log("=".repeat(68));
console.log("Listing agents using the paginateListAgents function:");
for (const agent of await listAgentsWithPaginator()) {
  console.log(agent);
}
}
```

- For API details, see [ListAgents](#) in *AWS SDK for JavaScript API Reference*.

Agents for Amazon Bedrock Runtime examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with Agents for Amazon Bedrock Runtime.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Topics

- [Actions](#)

Actions

InvokeAgent

The following code example shows how to use `InvokeAgent`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import {
  BedrockAgentRuntimeClient,
  InvokeAgentCommand,
} from "@aws-sdk/client-bedrock-agent-runtime";

/**
 * @typedef {Object} ResponseBody
 * @property {string} completion
 */

/**
 * Invokes a Bedrock agent to run an inference using the input
 * provided in the request body.
 *
 * @param {string} prompt - The prompt that you want the Agent to complete.
 * @param {string} sessionId - An arbitrary identifier for the session.
 */
export const invokeBedrockAgent = async (prompt, sessionId) => {
  const client = new BedrockAgentRuntimeClient({ region: "us-east-1" });
  // const client = new BedrockAgentRuntimeClient({
  //   region: "us-east-1",
  //   credentials: {
  //     accessKeyId: "accessKeyId", // permission to invoke agent
  //     secretAccessKey: "accessKeySecret",
  //   },
  // });

  const agentId = "AJBHXXILZN";
  const agentAliasId = "AVKP1ITZAA";

  const command = new InvokeAgentCommand({
    agentId,
    agentAliasId,
```

```
    sessionId,
    inputText: prompt,
  });

  try {
    let completion = "";
    const response = await client.send(command);

    if (response.completion === undefined) {
      throw new Error("Completion is undefined");
    }

    for await (let chunkEvent of response.completion) {
      const chunk = chunkEvent.chunk;
      console.log(chunk);
      const decodedResponse = new TextDecoder("utf-8").decode(chunk.bytes);
      completion += decodedResponse;
    }

    return { sessionId: sessionId, completion };
  } catch (err) {
    console.error(err);
  }
};

// Call function if run directly
import { fileURLToPath } from "url";
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  const result = await invokeBedrockAgent("I need help.", "123");
  console.log(result);
}
```

- For API details, see [InvokeAgent](#) in *AWS SDK for JavaScript API Reference*.

CloudWatch examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with CloudWatch.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Topics

- [Actions](#)

Actions

DeleteAlarms

The following code example shows how to use DeleteAlarms.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Import the SDK and client modules and call the API.

```
import { DeleteAlarmsCommand } from "@aws-sdk/client-cloudwatch";
import { client } from "../libs/client.js";

const run = async () => {
  const command = new DeleteAlarmsCommand({
    AlarmNames: [process.env.CLOUDWATCH_ALARM_NAME], // Set the value of
    CLOUDWATCH_ALARM_NAME to the name of an existing alarm.
  });

  try {
    return await client.send(command);
  } catch (err) {
    console.error(err);
  }
};

export default run();
```

Create the client in a separate module and export it.

```
import { CloudWatchClient } from "@aws-sdk/client-cloudwatch";

export const client = new CloudWatchClient({});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [DeleteAlarms](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Import the SDK and client modules and call the API.

```
// 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);
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).

- For API details, see [DeleteAlarms](#) in *AWS SDK for JavaScript API Reference*.

DescribeAlarmsForMetric

The following code example shows how to use `DescribeAlarmsForMetric`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Import the SDK and client modules and call the API.

```
import { DescribeAlarmsCommand } from "@aws-sdk/client-cloudwatch";
import { client } from "../libs/client.js";

const run = async () => {
  const command = new DescribeAlarmsCommand({
    AlarmNames: [process.env.CLOUDWATCH_ALARM_NAME], // Set the value of
    CLOUDWATCH_ALARM_NAME to the name of an existing alarm.
  });

  try {
    return await client.send(command);
  } catch (err) {
    console.error(err);
  }
};

export default run();
```

Create the client in a separate module and export it.

```
import { CloudWatchClient } from "@aws-sdk/client-cloudwatch";

export const client = new CloudWatchClient({});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [DescribeAlarmsForMetric](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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);
    });
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [DescribeAlarmsForMetric](#) in *AWS SDK for JavaScript API Reference*.

DisableAlarmActions

The following code example shows how to use `DisableAlarmActions`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Import the SDK and client modules and call the API.

```
import { DisableAlarmActionsCommand } from "@aws-sdk/client-cloudwatch";
import { client } from "../libs/client.js";

const run = async () => {
  const command = new DisableAlarmActionsCommand({
    AlarmNames: process.env.CLOUDWATCH_ALARM_NAME, // Set the value of
    CLOUDWATCH_ALARM_NAME to the name of an existing alarm.
  });

  try {
    return await client.send(command);
  } catch (err) {
    console.error(err);
  }
};

export default run();
```

Create the client in a separate module and export it.

```
import { CloudWatchClient } from "@aws-sdk/client-cloudwatch";

export const client = new CloudWatchClient({});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [DisableAlarmActions](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Import the SDK and client modules and call the API.

```
// 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);
    }
  }
);
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [DisableAlarmActions](#) in *AWS SDK for JavaScript API Reference*.

EnableAlarmActions

The following code example shows how to use `EnableAlarmActions`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Import the SDK and client modules and call the API.

```
import { EnableAlarmActionsCommand } from "@aws-sdk/client-cloudwatch";
import { client } from "../libs/client.js";

const run = async () => {
  const command = new EnableAlarmActionsCommand({
    AlarmNames: [process.env.CLOUDWATCH_ALARM_NAME], // Set the value of
    CLOUDWATCH_ALARM_NAME to the name of an existing alarm.
  });

  try {
    return await client.send(command);
  } catch (err) {
    console.error(err);
  }
};

export default run();
```

Create the client in a separate module and export it.

```
import { CloudWatchClient } from "@aws-sdk/client-cloudwatch";

export const client = new CloudWatchClient({});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [EnableAlarmActions](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Import the SDK and client modules and call the API.

```
// 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);
  }
});
}
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [EnableAlarmActions](#) in *AWS SDK for JavaScript API Reference*.

ListMetrics

The following code example shows how to use ListMetrics.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Import the SDK and client modules and call the API.

```
import { ListMetricsCommand } from "@aws-sdk/client-cloudwatch";
import { client } from "../libs/client.js";

export const main = () => {
  // Use the AWS console to see available namespaces and metric names. Custom
  // metrics can also be created.
  // https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/
  // viewing_metrics_with_cloudwatch.html
  const command = new ListMetricsCommand({
```

```
    Dimensions: [
      {
        Name: "LogGroupName",
      },
    ],
    MetricName: "IncomingLogEvents",
    Namespace: "AWS/Logs",
  });

  return client.send(command);
};
```

Create the client in a separate module and export it.

```
import { CloudWatchClient } from "@aws-sdk/client-cloudwatch";

export const client = new CloudWatchClient({});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [ListMetrics](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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));
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [ListMetrics](#) in *AWS SDK for JavaScript API Reference*.

PutMetricAlarm

The following code example shows how to use PutMetricAlarm.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Import the SDK and client modules and call the API.

```
import { PutMetricAlarmCommand } from "@aws-sdk/client-cloudwatch";
import { client } from "../libs/client.js";

const run = async () => {
  // This alarm triggers when CPUUtilization exceeds 70% for one minute.
  const command = new PutMetricAlarmCommand({
    AlarmName: process.env.CLOUDWATCH_ALARM_NAME, // Set the value of
    CLOUDWATCH_ALARM_NAME to the name of an existing alarm.
  });
```

```
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: process.env.EC2_INSTANCE_ID, // Set the value of EC_INSTANCE_ID to
the Id of an existing Amazon EC2 instance.
  },
],
Unit: "Percent",
});

try {
  return await client.send(command);
} catch (err) {
  console.error(err);
}
};

export default run();
```

Create the client in a separate module and export it.

```
import { CloudWatchClient } from "@aws-sdk/client-cloudwatch";

export const client = new CloudWatchClient({});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [PutMetricAlarm](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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);
  }
});
```

```
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [PutMetricAlarm](#) in *AWS SDK for JavaScript API Reference*.

PutMetricData

The following code example shows how to use PutMetricData.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Import the SDK and client modules and call the API.

```
import { PutMetricDataCommand } from "@aws-sdk/client-cloudwatch";
import { client } from "../libs/client.js";

const run = async () => {
  // See https://docs.aws.amazon.com/AmazonCloudWatch/latest/APIReference/
  // API_PutMetricData.html#API_PutMetricData_RequestParameters
  // and https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/
  // publishingMetrics.html
  // for more information about the parameters in this command.
  const command = new PutMetricDataCommand({
    MetricData: [
      {
        MetricName: "PAGES_VISITED",
        Dimensions: [
          {
            Name: "UNIQUE_PAGES",
            Value: "URLS",
          },
        ],
        Unit: "None",
        Value: 1.0,
      },
    ],
  });
```

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

try {
  return await client.send(command);
} catch (err) {
  console.error(err);
}
};

export default run();
```

Create the client in a separate module and export it.

```
import { CloudWatchClient } from "@aws-sdk/client-cloudwatch";

export const client = new CloudWatchClient({});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [PutMetricData](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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));
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [PutMetricData](#) in *AWS SDK for JavaScript API Reference*.

CloudWatch Events examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with CloudWatch Events.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Topics

- [Actions](#)

Actions

PutEvents

The following code example shows how to use PutEvents.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Import the SDK and client modules and call the API.

```
import { PutEventsCommand } from "@aws-sdk/client-cloudwatch-events";
import { client } from "../libs/client.js";

const run = async () => {
  const command = new PutEventsCommand({
    // The list of events to send to Amazon CloudWatch Events.
    Entries: [
      {
        // The name of the application or service that is sending the event.
        Source: "my.app",

        // The name of the event that is being sent.
        DetailType: "My Custom Event",

        // The data that is sent with the event.
        Detail: JSON.stringify({ timeOfEvent: new Date().toISOString() }),
      },
    ],
  });

  try {
    return await client.send(command);
  } catch (err) {
```

```
    console.error(err);
  }
};

export default run();
```

Create the client in a separate module and export it.

```
import { CloudWatchEventsClient } from "@aws-sdk/client-cloudwatch-events";

export const client = new CloudWatchEventsClient({});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [PutEvents](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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);
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [PutEvents](#) in *AWS SDK for JavaScript API Reference*.

PutRule

The following code example shows how to use PutRule.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Import the SDK and client modules and call the API.

```
import { PutRuleCommand } from "@aws-sdk/client-cloudwatch-events";
import { client } from "../libs/client.js";

const run = async () => {
  // Request parameters for PutRule.
  // https://docs.aws.amazon.com/eventbridge/latest/APIReference/API_PutRule.html#API_PutRule_RequestParameters
  const command = new PutRuleCommand({
    Name: process.env.CLOUDWATCH_EVENTS_RULE,

    // The event pattern for the rule.
    // Example: {"source": ["my.app"]}
    EventPattern: process.env.CLOUDWATCH_EVENTS_RULE_PATTERN,
```

```
// The state of the rule. Valid values: ENABLED, DISABLED
State: "ENABLED",
});

try {
  return await client.send(command);
} catch (err) {
  console.error(err);
}
};

export default run();
```

Create the client in a separate module and export it.

```
import { CloudWatchEventsClient } from "@aws-sdk/client-cloudwatch-events";

export const client = new CloudWatchEventsClient({});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [PutRule](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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);
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [PutRule](#) in *AWS SDK for JavaScript API Reference*.

PutTargets

The following code example shows how to use PutTargets.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Import the SDK and client modules and call the API.

```
import { PutTargetsCommand } from "@aws-sdk/client-cloudwatch-events";
import { client } from "../libs/client.js";

const run = async () => {
  const command = new PutTargetsCommand({
    // The name of the Amazon CloudWatch Events rule.
    Rule: process.env.CLOUDWATCH_EVENTS_RULE,

    // The targets to add to the rule.
```

```
Targets: [  
  {  
    Arn: process.env.CLOUDWATCH_EVENTS_TARGET_ARN,  
    // The ID of the target. Choose a unique ID for each target.  
    Id: process.env.CLOUDWATCH_EVENTS_TARGET_ID,  
  },  
],  
});  
  
try {  
  return await client.send(command);  
} catch (err) {  
  console.error(err);  
}  
};  
  
export default run();
```

Create the client in a separate module and export it.

```
import { CloudWatchEventsClient } from "@aws-sdk/client-cloudwatch-events";  
  
export const client = new CloudWatchEventsClient({});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [PutTargets](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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);
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [PutTargets](#) in *AWS SDK for JavaScript API Reference*.

CloudWatch Logs examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with CloudWatch Logs.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Scenarios are code examples that show you how to accomplish specific tasks by calling multiple functions within a service or combined with other AWS services.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Topics

- [Actions](#)

- [Scenarios](#)

Actions

CreateLogGroup

The following code example shows how to use CreateLogGroup.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { CreateLogGroupCommand } from "@aws-sdk/client-cloudwatch-logs";
import { client } from "../libs/client.js";

const run = async () => {
  const command = new CreateLogGroupCommand({
    // The name of the log group.
    logGroupName: process.env.CLOUDWATCH_LOGS_LOG_GROUP,
  });

  try {
    return await client.send(command);
  } catch (err) {
    console.error(err);
  }
};

export default run();
```

- For API details, see [CreateLogGroup](#) in *AWS SDK for JavaScript API Reference*.

DeleteLogGroup

The following code example shows how to use DeleteLogGroup.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { DeleteLogGroupCommand } from "@aws-sdk/client-cloudwatch-logs";
import { client } from "../libs/client.js";

const run = async () => {
  const command = new DeleteLogGroupCommand({
    // The name of the log group.
    logGroupName: process.env.CLOUDWATCH_LOGS_LOG_GROUP,
  });

  try {
    return await client.send(command);
  } catch (err) {
    console.error(err);
  }
};

export default run();
```

- For API details, see [DeleteLogGroup](#) in *AWS SDK for JavaScript API Reference*.

DeleteSubscriptionFilter

The following code example shows how to use DeleteSubscriptionFilter.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { DeleteSubscriptionFilterCommand } from "@aws-sdk/client-cloudwatch-logs";
import { client } from "../libs/client.js";

const run = async () => {
  const command = new DeleteSubscriptionFilterCommand({
    // The name of the filter.
    filterName: process.env.CLOUDWATCH_LOGS_FILTER_NAME,
    // The name of the log group.
    logGroupName: process.env.CLOUDWATCH_LOGS_LOG_GROUP,
  });

  try {
    return await client.send(command);
  } catch (err) {
    console.error(err);
  }
};

export default run();
```

- For API details, see [DeleteSubscriptionFilter](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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);
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [DeleteSubscriptionFilter](#) in *AWS SDK for JavaScript API Reference*.

DescribeLogGroups

The following code example shows how to use DescribeLogGroups.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import {
  paginateDescribeLogGroups,
  CloudWatchLogsClient,
} from "@aws-sdk/client-cloudwatch-logs";

const client = new CloudWatchLogsClient({});

export const main = async () => {
  const paginatedLogGroups = paginateDescribeLogGroups({ client }, {});
  const logGroups = [];

  for await (const page of paginatedLogGroups) {
    if (page.logGroups && page.logGroups.every((lg) => !!lg)) {
      logGroups.push(...page.logGroups);
    }
  }
}
```

```
    }  
  
    console.log(logGroups);  
    return logGroups;  
  };
```

- For API details, see [DescribeLogGroups](#) in *AWS SDK for JavaScript API Reference*.

DescribeSubscriptionFilters

The following code example shows how to use `DescribeSubscriptionFilters`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { DescribeSubscriptionFiltersCommand } from "@aws-sdk/client-cloudwatch-logs";  
import { client } from "../libs/client.js";  
  
const run = async () => {  
  // This will return a list of all subscription filters in your account  
  // matching the log group name.  
  const command = new DescribeSubscriptionFiltersCommand({  
    logGroupName: process.env.CLOUDWATCH_LOGS_LOG_GROUP,  
    limit: 1,  
  });  
  
  try {  
    return await client.send(command);  
  } catch (err) {  
    console.error(err);  
  }  
};  
  
export default run();
```


- For API details, see [DescribeSubscriptionFilters](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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 cw1 = new AWS.CloudWatchLogs({ apiVersion: "2014-03-28" });

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

cw1.describeSubscriptionFilters(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.subscriptionFilters);
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [DescribeSubscriptionFilters](#) in *AWS SDK for JavaScript API Reference*.

GetQueryResults

The following code example shows how to use `GetQueryResults`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
/**
 * Simple wrapper for the GetQueryResultsCommand.
 * @param {string} queryId
 */
_getQueryResults(queryId) {
  return this.client.send(new GetQueryResultsCommand({ queryId }));
}
```

- For API details, see [GetQueryResults](#) in *AWS SDK for JavaScript API Reference*.

PutSubscriptionFilter

The following code example shows how to use PutSubscriptionFilter.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { PutSubscriptionFilterCommand } from "@aws-sdk/client-cloudwatch-logs";
import { client } from "../libs/client.js";

const run = async () => {
  const command = new PutSubscriptionFilterCommand({
    // An ARN of a same-account Kinesis stream, Kinesis Firehose
    // delivery stream, or Lambda function.
    // https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/
    SubscriptionFilters.html
  });
```

```
destinationArn: process.env.CLOUDWATCH_LOGS_DESTINATION_ARN,

// A name for the filter.
filterName: process.env.CLOUDWATCH_LOGS_FILTER_NAME,

// A filter pattern for subscribing to a filtered stream of log events.
// https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/
FilterAndPatternSyntax.html
filterPattern: process.env.CLOUDWATCH_LOGS_FILTER_PATTERN,

// The name of the log group. Messages in this group matching the filter pattern
// will be sent to the destination ARN.
logGroupName: process.env.CLOUDWATCH_LOGS_LOG_GROUP,
});

try {
  return await client.send(command);
} catch (err) {
  console.error(err);
}
};

export default run();
```

- For API details, see [PutSubscriptionFilter](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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 cw1 = 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);
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [PutSubscriptionFilter](#) in *AWS SDK for JavaScript API Reference*.

StartLiveTail

The following code example shows how to use StartLiveTail.

SDK for JavaScript (v3)

Include the required files.

```
import { CloudWatchLogsClient, StartLiveTailCommand } from "@aws-sdk/client-cloudwatch-logs";
```

Handle the events from the Live Tail session.

```
async function handleResponseAsync(response) {
  try {
    for await (const event of response.responseStream) {
      if (event.sessionStart !== undefined) {
        console.log(event.sessionStart);
      } else if (event.sessionUpdate !== undefined) {
        for (const logEvent of event.sessionUpdate.sessionResults) {
          const timestamp = logEvent.timestamp;
        }
      }
    }
  }
}
```

```
        const date = new Date(timestamp);
        console.log("[ " + date + " ] " + logEvent.message);
    }
    } else {
        console.error("Unknown event type");
    }
}
} catch (err) {
    // On-stream exceptions are captured here
    console.error(err)
}
}
```

Start the Live Tail session.

```
const client = new CloudWatchLogsClient();

const command = new StartLiveTailCommand({
    logGroupIdentifiers: logGroupIdentifiers,
    logStreamNames: logStreamNames,
    logEventFilterPattern: filterPattern
});
try{
    const response = await client.send(command);
    handleResponseAsync(response);
} catch (err){
    // Pre-stream exceptions are captured here
    console.log(err);
}
```

Stop the Live Tail session after a period of time has elapsed.

```
/* Set a timeout to close the client. This will stop the Live Tail session. */
setTimeout(function() {
    console.log("Client timeout");
    client.destroy();
}, 10000);
```

- For API details, see [StartLiveTail](#) in *AWS SDK for JavaScript API Reference*.

StartQuery

The following code example shows how to use StartQuery.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
/**
 * Wrapper for the StartQueryCommand. Uses a static query string
 * for consistency.
 * @param {[Date, Date]} dateRange
 * @param {number} maxLogs
 * @returns {Promise<{ queryId: string }>}
 */
async _startQuery([startDate, endDate], maxLogs = 10000) {
  try {
    return await this.client.send(
      new StartQueryCommand({
        logGroupNames: this.logGroupNames,
        queryString: "fields @timestamp, @message | sort @timestamp asc",
        startTime: startDate.valueOf(),
        endTime: endDate.valueOf(),
        limit: maxLogs,
      }),
    );
  } catch (err) {
    /** @type {string} */
    const message = err.message;
    if (message.startsWith("Query's end date and time")) {
      // This error indicates that the query's start or end date occur
      // before the log group was created.
      throw new DateOutOfBoundsError(message);
    }

    throw err;
  }
}
```

- For API details, see [StartQuery](#) in *AWS SDK for JavaScript API Reference*.

Scenarios

Run a large query

The following code example shows how to use CloudWatch Logs to query more than 10,000 records.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

This is the entry point.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { CloudWatchLogsClient } from "@aws-sdk/client-cloudwatch-logs";
import { CloudWatchQuery } from "./cloud-watch-query.js";

console.log("Starting a recursive query...");

if (!process.env.QUERY_START_DATE || !process.env.QUERY_END_DATE) {
  throw new Error(
    "QUERY_START_DATE and QUERY_END_DATE environment variables are required.",
  );
}

const cloudWatchQuery = new CloudWatchQuery(new CloudWatchLogsClient({}), {
  logGroupNames: ["/workflows/cloudwatch-logs/large-query"],
  dateRange: [
    new Date(parseInt(process.env.QUERY_START_DATE)),
    new Date(parseInt(process.env.QUERY_END_DATE)),
  ],
});
```

```
await cloudWatchQuery.run();

console.log(
  `Queries finished in ${cloudWatchQuery.secondsElapsed} seconds.\nTotal logs found:
  ${cloudWatchQuery.results.length}`,
);
```

This is a class that splits queries into multiple steps if necessary.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import {
  StartQueryCommand,
  GetQueryResultsCommand,
} from "@aws-sdk/client-cloudwatch-logs";
import { splitDateRange } from "@aws-doc-sdk-examples/lib/utils/util-date.js";
import { retry } from "@aws-doc-sdk-examples/lib/utils/util-timers.js";

class DateOutOfBoundsError extends Error {}

export class CloudWatchQuery {
  /**
   * Run a query for all CloudWatch Logs within a certain date range.
   * CloudWatch logs return a max of 10,000 results. This class
   * performs a binary search across all of the logs in the provided
   * date range if a query returns the maximum number of results.
   *
   * @param {import('@aws-sdk/client-cloudwatch-logs').CloudWatchLogsClient} client
   * @param {{ logGroupNames: string[], dateRange: [Date, Date], queryConfig:
   { limit: number } }} config
   */
  constructor(client, { logGroupNames, dateRange, queryConfig }) {
    this.client = client;
    /**
     * All log groups are queried.
     */
    this.logGroupNames = logGroupNames;

    /**
     * The inclusive date range that is queried.
     */
  }
}
```



```

    this.dateRange = dateRange;

    /**
     * CloudWatch Logs never returns more than 10,000 logs.
     */
    this.limit = queryConfig?.limit ?? 10000;

    /**
     * @type {import("@aws-sdk/client-cloudwatch-logs").ResultField[][]}
     */
    this.results = [];
  }

  /**
   * Run the query.
   */
  async run() {
    this.secondsElapsed = 0;
    const start = new Date();
    this.results = await this._largeQuery(this.dateRange);
    const end = new Date();
    this.secondsElapsed = (end - start) / 1000;
    return this.results;
  }

  /**
   * Recursively query for logs.
   * @param {[Date, Date]} dateRange
   * @returns {Promise<import("@aws-sdk/client-cloudwatch-logs").ResultField[][]>}
   */
  async _largeQuery(dateRange) {
    const logs = await this._query(dateRange, this.limit);

    console.log(
      `Query date range: ${dateRange
        .map((d) => d.toISOString())
        .join(" to ")}. Found ${logs.length} logs.`
    );

    if (logs.length < this.limit) {
      return logs;
    }

    const lastLogDate = this._getLastLogDate(logs);

```

```

    const offsetLastLogDate = new Date(lastLogDate);
    offsetLastLogDate.setMilliseconds(lastLogDate.getMilliseconds() + 1);
    const subDateRange = [offsetLastLogDate, dateRange[1]];
    const [r1, r2] = splitDateRange(subDateRange);
    const results = await Promise.all([
      this._largeQuery(r1),
      this._largeQuery(r2),
    ]);
    return [logs, ...results].flat();
  }

  /**
   * Find the most recent log in a list of logs.
   * @param {import("@aws-sdk/client-cloudwatch-logs").ResultField[][]} logs
   */
  _getLastLogDate(logs) {
    const timestamps = logs
      .map(
        (log) =>
          log.find((fieldMeta) => fieldMeta.field === "@timestamp")?.value,
      )
      .filter((t) => !!t)
      .map((t) => `${t}Z`)
      .sort();

    if (!timestamps.length) {
      throw new Error("No timestamp found in logs.");
    }

    return new Date(timestamps[timestamps.length - 1]);
  }

  /**
   * Simple wrapper for the GetQueryResultsCommand.
   * @param {string} queryId
   */
  _getQueryResults(queryId) {
    return this.client.send(new GetQueryResultsCommand({ queryId }));
  }

  /**
   * Starts a query and waits for it to complete.
   * @param {[Date, Date]} dateRange
   * @param {number} maxLogs

```

```
*/
async _query(dateRange, maxLogs) {
  try {
    const { queryId } = await this._startQuery(dateRange, maxLogs);
    const { results } = await this._waitUntilQueryDone(queryId);
    return results ?? [];
  } catch (err) {
    /**
     * This error is thrown when StartQuery returns an error indicating
     * that the query's start or end date occur before the log group was
     * created.
     */
    if (err instanceof DateOutOfBoundsError) {
      return [];
    } else {
      throw err;
    }
  }
}

/**
 * Wrapper for the StartQueryCommand. Uses a static query string
 * for consistency.
 * @param {[Date, Date]} dateRange
 * @param {number} maxLogs
 * @returns {Promise<{ queryId: string }>}
 */
async _startQuery([startDate, endDate], maxLogs = 10000) {
  try {
    return await this.client.send(
      new StartQueryCommand({
        logGroupNames: this.logGroupNames,
        queryString: "fields @timestamp, @message | sort @timestamp asc",
        startTime: startDate.valueOf(),
        endTime: endDate.valueOf(),
        limit: maxLogs,
      }),
    );
  } catch (err) {
    /** @type {string} */
    const message = err.message;
    if (message.startsWith("Query's end date and time")) {
      // This error indicates that the query's start or end date occur
      // before the log group was created.
    }
  }
}
```

```
        throw new DateOutOfBoundsError(message);
    }

    throw err;
}

/**
 * Call GetQueryResultsCommand until the query is done.
 * @param {string} queryId
 */
_waitUntilQueryDone(queryId) {
    const getResults = async () => {
        const results = await this._getQueryResults(queryId);
        const queryDone = [
            "Complete",
            "Failed",
            "Cancelled",
            "Timeout",
            "Unknown",
        ].includes(results.status);

        return { queryDone, results };
    };

    return retry(
        { intervalInMs: 1000, maxRetries: 60, quiet: true },
        async () => {
            const { queryDone, results } = await getResults();
            if (!queryDone) {
                throw new Error("Query not done.");
            }

            return results;
        },
    );
}
}
```

- For API details, see the following topics in *AWS SDK for JavaScript API Reference*.
 - [GetQueryResults](#)
 - [StartQuery](#)

CodeBuild examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with CodeBuild.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Topics

- [Actions](#)

Actions

CreateProject

The following code example shows how to use CreateProject.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create a project.

```
import {
  ArtifactsType,
  CodeBuildClient,
  ComputeType,
  CreateProjectCommand,
  EnvironmentType,
  SourceType,
} from "@aws-sdk/client-codebuild";

// Create the AWS CodeBuild project.
export const createProject = async (
```

```
projectName = "MyCodeBuilder",
roleArn = "arn:aws:iam::xxxxxxxxxxxx:role/CodeBuildAdmin",
buildOutputBucket = "xxxx",
githubUrl = "https://...",
) => {
  const codeBuildClient = new CodeBuildClient({});

  const response = await codeBuildClient.send(
    new CreateProjectCommand({
      artifacts: {
        // The destination of the build artifacts.
        type: ArtifactsType.S3,
        location: buildOutputBucket,
      },
      // Information about the build environment. The combination of "computeType"
and "type" determines the
      // requirements for the environment such as CPU, memory, and disk space.
      environment: {
        // Build environment compute types.
        // https://docs.aws.amazon.com/codebuild/latest/userguide/build-env-ref-
compute-types.html
        computeType: ComputeType.BUILD_GENERAL1_SMALL,
        // Docker image identifier.
        // See https://docs.aws.amazon.com/codebuild/latest/userguide/build-env-ref-
available.html
        image: "aws/codebuild/standard:7.0",
        // Build environment type.
        type: EnvironmentType.LINUX_CONTAINER,
      },
      name: projectName,
      // A role ARN with permission to create a CodeBuild project, write to the
artifact location, and write CloudWatch logs.
      serviceRole: roleArn,
      source: {
        // The type of repository that contains the source code to be built.
        type: SourceType.GITHUB,
        // The location of the repository that contains the source code to be built.
        location: githubUrl,
      },
    }
  ),
);
console.log(response);
// {
//   '$metadata': {
```

```
//     statusCode: 200,
//     requestId: 'b428b244-777b-49a6-a48d-5dffedced8e7',
//     extendedRequestId: undefined,
//     cfId: undefined,
//     attempts: 1,
//     totalRetryDelay: 0
//   },
//   project: {
//     arn: 'arn:aws:codebuild:us-east-1:xxxxxxxxxxxx:project/MyCodeBuilder',
//     artifacts: {
//       encryptionDisabled: false,
//       location: 'xxxxxx-xxxxxx-xxxxxx',
//       name: 'MyCodeBuilder',
//       namespaceType: 'NONE',
//       packaging: 'NONE',
//       type: 'S3'
//     },
//     badge: { badgeEnabled: false },
//     cache: { type: 'NO_CACHE' },
//     created: 2023-08-18T14:46:48.979Z,
//     encryptionKey: 'arn:aws:kms:us-east-1:xxxxxxxxxxxx:alias/aws/s3',
//     environment: {
//       computeType: 'BUILD_GENERAL1_SMALL',
//       environmentVariables: [],
//       image: 'aws/codebuild/standard:7.0',
//       imagePullCredentialsType: 'CODEBUILD',
//       privilegedMode: false,
//       type: 'LINUX_CONTAINER'
//     },
//     lastModified: 2023-08-18T14:46:48.979Z,
//     name: 'MyCodeBuilder',
//     projectVisibility: 'PRIVATE',
//     queuedTimeoutInMinutes: 480,
//     serviceRole: 'arn:aws:iam::xxxxxxxxxxxx:role/CodeBuildAdmin',
//     source: {
//       insecureSsl: false,
//       location: 'https://...',
//       reportBuildStatus: false,
//       type: 'GITHUB'
//     },
//     timeoutInMinutes: 60
//   }
// }
return response;
```

```
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [CreateProject](#) in *AWS SDK for JavaScript API Reference*.

Amazon Cognito Identity Provider examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with Amazon Cognito Identity Provider.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Scenarios are code examples that show you how to accomplish specific tasks by calling multiple functions within a service or combined with other AWS services.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Get started

Hello Amazon Cognito

The following code examples show how to get started using Amazon Cognito.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import {
  paginateListUserPools,
  CognitoIdentityProviderClient,
} from "@aws-sdk/client-cognito-identity-provider";
```



```
const client = new CognitoIdentityProviderClient({});

export const helloCognito = async () => {
  const paginator = paginateListUserPools({ client }, {});

  const userPoolNames = [];

  for await (const page of paginator) {
    const names = page.UserPools.map((pool) => pool.Name);
    userPoolNames.push(...names);
  }

  console.log("User pool names: ");
  console.log(userPoolNames.join("\n"));
  return userPoolNames;
};
```

- For API details, see [ListUserPools](#) in *AWS SDK for JavaScript API Reference*.

Topics

- [Actions](#)
- [Scenarios](#)

Actions

AdminGetUser

The following code example shows how to use AdminGetUser.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const adminGetUser = ({ userPoolId, username }) => {
```

```
const client = new CognitoIdentityProviderClient({});

const command = new AdminGetUserCommand({
  UserPoolId: userPoolId,
  Username: username,
});

return client.send(command);
};
```

- For API details, see [AdminGetUser](#) in *AWS SDK for JavaScript API Reference*.

AdminInitiateAuth

The following code example shows how to use AdminInitiateAuth.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const adminInitiateAuth = ({ clientId, userPoolId, username, password }) => {
  const client = new CognitoIdentityProviderClient({});

  const command = new AdminInitiateAuthCommand({
    ClientId: clientId,
    UserPoolId: userPoolId,
    AuthFlow: AuthFlowType.ADMIN_USER_PASSWORD_AUTH,
    AuthParameters: { USERNAME: username, PASSWORD: password },
  });

  return client.send(command);
};
```

- For API details, see [AdminInitiateAuth](#) in *AWS SDK for JavaScript API Reference*.

AdminRespondToAuthChallenge

The following code example shows how to use `AdminRespondToAuthChallenge`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const adminRespondToAuthChallenge = ({
  userPoolId,
  clientId,
  username,
  totp,
  session,
}) => {
  const client = new CognitoIdentityProviderClient({});
  const command = new AdminRespondToAuthChallengeCommand({
    ChallengeName: ChallengeNameType.SOFTWARE_TOKEN_MFA,
    ChallengeResponses: {
      SOFTWARE_TOKEN_MFA_CODE: totp,
      USERNAME: username,
    },
    ClientId: clientId,
    UserPoolId: userPoolId,
    Session: session,
  });

  return client.send(command);
};
```

- For API details, see [AdminRespondToAuthChallenge](#) in *AWS SDK for JavaScript API Reference*.

AssociateSoftwareToken

The following code example shows how to use `AssociateSoftwareToken`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const associateSoftwareToken = (session) => {
  const client = new CognitoIdentityProviderClient({});
  const command = new AssociateSoftwareTokenCommand({
    Session: session,
  });

  return client.send(command);
};
```

- For API details, see [AssociateSoftwareToken](#) in *AWS SDK for JavaScript API Reference*.

ConfirmDevice

The following code example shows how to use ConfirmDevice.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const confirmDevice = ({ deviceKey, accessToken, passwordVerifier, salt }) => {
  const client = new CognitoIdentityProviderClient({});

  const command = new ConfirmDeviceCommand({
    DeviceKey: deviceKey,
    AccessToken: accessToken,
    DeviceSecretVerifierConfig: {
      PasswordVerifier: passwordVerifier,
    }
  });
```

```
        Salt: salt,
      },
    });

    return client.send(command);
  };
```

- For API details, see [ConfirmDevice](#) in *AWS SDK for JavaScript API Reference*.

ConfirmSignUp

The following code example shows how to use `ConfirmSignUp`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const confirmSignUp = ({ clientId, username, code }) => {
  const client = new CognitoIdentityProviderClient({});

  const command = new ConfirmSignUpCommand({
    ClientId: clientId,
    Username: username,
    ConfirmationCode: code,
  });

  return client.send(command);
};
```

- For API details, see [ConfirmSignUp](#) in *AWS SDK for JavaScript API Reference*.

InitiateAuth

The following code example shows how to use `InitiateAuth`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const initiateAuth = ({ username, password, clientId }) => {
  const client = new CognitoIdentityProviderClient({});

  const command = new InitiateAuthCommand({
    AuthFlow: AuthFlowType.USER_PASSWORD_AUTH,
    AuthParameters: {
      USERNAME: username,
      PASSWORD: password,
    },
    ClientId: clientId,
  });

  return client.send(command);
};
```

- For API details, see [InitiateAuth](#) in *AWS SDK for JavaScript API Reference*.

ListUsers

The following code example shows how to use ListUsers.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const listUsers = ({ userPoolId }) => {
  const client = new CognitoIdentityProviderClient({});
```

```
const command = new ListUsersCommand({
  UserPoolId: userPoolId,
});

return client.send(command);
};
```

- For API details, see [ListUsers](#) in *AWS SDK for JavaScript API Reference*.

ResendConfirmationCode

The following code example shows how to use ResendConfirmationCode.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const resendConfirmationCode = ({ clientId, username }) => {
  const client = new CognitoIdentityProviderClient({});

  const command = new ResendConfirmationCodeCommand({
    ClientId: clientId,
    Username: username,
  });

  return client.send(command);
};
```

- For API details, see [ResendConfirmationCode](#) in *AWS SDK for JavaScript API Reference*.

RespondToAuthChallenge

The following code example shows how to use RespondToAuthChallenge.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const respondToAuthChallenge = ({
  clientId,
  username,
  session,
  userPoolId,
  code,
}) => {
  const client = new CognitoIdentityProviderClient({});

  const command = new RespondToAuthChallengeCommand({
    ChallengeName: ChallengeNameType.SOFTWARE_TOKEN_MFA,
    ChallengeResponses: {
      SOFTWARE_TOKEN_MFA_CODE: code,
      USERNAME: username,
    },
    ClientId: clientId,
    UserPoolId: userPoolId,
    Session: session,
  });

  return client.send(command);
};
```

- For API details, see [RespondToAuthChallenge](#) in *AWS SDK for JavaScript API Reference*.

SignUp

The following code example shows how to use SignUp.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const signUp = ({ clientId, username, password, email }) => {
  const client = new CognitoIdentityProviderClient({});

  const command = new SignUpCommand({
    ClientId: clientId,
    Username: username,
    Password: password,
    UserAttributes: [{ Name: "email", Value: email }],
  });

  return client.send(command);
};
```

- For API details, see [SignUp](#) in *AWS SDK for JavaScript API Reference*.

VerifySoftwareToken

The following code example shows how to use `VerifySoftwareToken`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const verifySoftwareToken = (totp) => {
  const client = new CognitoIdentityProviderClient({});

  // The 'Session' is provided in the response to 'AssociateSoftwareToken'.
```

```
const session = process.env.SESSION;

if (!session) {
  throw new Error(
    "Missing a valid Session. Did you run 'admin-initiate-auth'?",
  );
}

const command = new VerifySoftwareTokenCommand({
  Session: session,
  UserCode: totp,
});

return client.send(command);
};
```

- For API details, see [VerifySoftwareToken](#) in *AWS SDK for JavaScript API Reference*.

Scenarios

Sign up a user with a user pool that requires MFA

The following code example shows how to:

- Sign up and confirm a user with a username, password, and email address.
- Set up multi-factor authentication by associating an MFA application with the user.
- Sign in by using a password and an MFA code.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

For the best experience, clone the GitHub repository and run this example. The following code represents a sample of the full example application.

```
import { log } from "@aws-doc-sdk-examples/lib/utils/util-log.js";
import { signUp } from "../../actions/sign-up.js";
import { FILE_USER_POOLS } from "./constants.js";
import { getSecondValuesFromEntries } from "@aws-doc-sdk-examples/lib/utils/util-csv.js";

const validateClient = (clientId) => {
  if (!clientId) {
    throw new Error(
      `App client id is missing. Did you run 'create-user-pool'?`,
    );
  }
};

const validateUser = (username, password, email) => {
  if (!(username && password && email)) {
    throw new Error(
      `Username, password, and email must be provided as arguments to the 'sign-up' command.`,
    );
  }
};

const signUpHandler = async (commands) => {
  const [, username, password, email] = commands;

  try {
    validateUser(username, password, email);
    /**
     * @type {string[]}
     */
    const values = getSecondValuesFromEntries(FILE_USER_POOLS);
    const clientId = values[0];
    validateClient(clientId);
    log(`Signing up.`);
    await signUp({ clientId, username, password, email });
    log(`Signed up. A confirmation email has been sent to: ${email}.`);
    log(`Run 'confirm-sign-up ${username} <code>' to confirm your account.`);
  } catch (err) {
    log(err);
  }
};
```

```
export { signUpHandler };

const signUp = ({ clientId, username, password, email }) => {
  const client = new CognitoIdentityProviderClient({});

  const command = new SignUpCommand({
    ClientId: clientId,
    Username: username,
    Password: password,
    UserAttributes: [{ Name: "email", Value: email }],
  });

  return client.send(command);
};

import { log } from "@aws-doc-sdk-examples/lib/utils/util-log.js";
import { confirmSignUp } from "../../actions/confirm-sign-up.js";
import { FILE_USER_POOLS } from "./constants.js";
import { getSecondValuesFromEntries } from "@aws-doc-sdk-examples/lib/utils/util-csv.js";

const validateClient = (clientId) => {
  if (!clientId) {
    throw new Error(
      `App client id is missing. Did you run 'create-user-pool'?`,
    );
  }
};

const validateUser = (username) => {
  if (!username) {
    throw new Error(
      `Username name is missing. It must be provided as an argument to the 'confirm-sign-up' command.`,
    );
  }
};

const validateCode = (code) => {
  if (!code) {
    throw new Error(
      `Verification code is missing. It must be provided as an argument to the 'confirm-sign-up' command.`,
    );
  }
};
```

```
    }
  };

const confirmSignUpHandler = async (commands) => {
  const [_ , username, code] = commands;

  try {
    validateUser(username);
    validateCode(code);
    /**
     * @type {string[]}
     */
    const values = getSecondValuesFromEntries(FILE_USER_POOLS);
    const clientId = values[0];
    validateClient(clientId);
    log(`Confirming user.`);
    await confirmSignUp({ clientId, username, code });
    log(
      `User confirmed. Run 'admin-initiate-auth ${username} <password>' to sign
in.` ,
    );
  } catch (err) {
    log(err);
  }
};

export { confirmSignUpHandler };

const confirmSignUp = ({ clientId, username, code }) => {
  const client = new CognitoIdentityProviderClient({});

  const command = new ConfirmSignUpCommand({
    ClientId: clientId,
    Username: username,
    ConfirmationCode: code,
  });

  return client.send(command);
};

import qrCode from "qr-code-terminal";
import { log } from "@aws-doc-sdk-examples/lib/utils/util-log.js";
import { adminInitiateAuth } from "../../actions/admin-initiate-auth.js";
```

```
import { associateSoftwareToken } from "../../../../../actions/associate-software-token.js";
import { FILE_USER_POOLS } from "../constants.js";
import { getFirstEntry } from "@aws-doc-sdk-examples/lib/utils/util-csv.js";

const handleMfaSetup = async (session, username) => {
  const { SecretCode, Session } = await associateSoftwareToken(session);

  // Store the Session for use with 'VerifySoftwareToken'.
  process.env.SESSION = Session;

  console.log(
    "Scan this code in your preferred authenticator app, then run 'verify-software-token' to finish the setup.",
  );
  qrcode.generate(
    `otpauth://totp/${username}?secret=${SecretCode}`,
    { small: true },
    console.log,
  );
};

const handleSoftwareTokenMfa = (session) => {
  // Store the Session for use with 'AdminRespondToAuthChallenge'.
  process.env.SESSION = session;
};

const validateClient = (id) => {
  if (!id) {
    throw new Error(
      `User pool client id is missing. Did you run 'create-user-pool'?`,
    );
  }
};

const validateId = (id) => {
  if (!id) {
    throw new Error(`User pool id is missing. Did you run 'create-user-pool'?`);
  }
};

const validateUser = (username, password) => {
  if (!(username && password)) {
    throw new Error(
```

```
    `Username and password must be provided as arguments to the 'admin-initiate-
auth' command.` ,
  );
}
};

const adminInitiateAuthHandler = async (commands) => {
  const [, username, password] = commands;

  try {
    validateUser(username, password);

    const [userPoolId, clientId] = getFirstEntry(FILE_USER_POOLS);
    validateId(userPoolId);
    validateClient(clientId);

    log("Signing in.");
    const { ChallengeName, Session } = await adminInitiateAuth({
      clientId,
      userPoolId,
      username,
      password,
    });

    if (ChallengeName === "MFA_SETUP") {
      log("MFA setup is required.");
      return handleMfaSetup(Session, username);
    }

    if (ChallengeName === "SOFTWARE_TOKEN_MFA") {
      handleSoftwareTokenMfa(Session);
      log(`Run 'admin-respond-to-auth-challenge ${username} <totp>'`);
    }
  } catch (err) {
    log(err);
  }
};

export { adminInitiateAuthHandler };

const adminInitiateAuth = ({ clientId, userPoolId, username, password }) => {
  const client = new CognitoIdentityProviderClient({});

  const command = new AdminInitiateAuthCommand({
```

```
    ClientId: clientId,
    UserPoolId: userPoolId,
    AuthFlow: AuthFlowType.ADMIN_USER_PASSWORD_AUTH,
    AuthParameters: { USERNAME: username, PASSWORD: password },
  });

  return client.send(command);
};

import { log } from "@aws-doc-sdk-examples/lib/utils/util-log.js";
import { adminRespondToAuthChallenge } from "../../actions/admin-respond-to-auth-challenge.js";
import { getFirstEntry } from "@aws-doc-sdk-examples/lib/utils/util-csv.js";
import { FILE_USER_POOLS } from "./constants.js";

const verifyUsername = (username) => {
  if (!username) {
    throw new Error(
      `Username is missing. It must be provided as an argument to the 'admin-respond-to-auth-challenge' command.`
    );
  }
};

const verifyTotp = (totp) => {
  if (!totp) {
    throw new Error(
      `Time-based one-time password (TOTP) is missing. It must be provided as an argument to the 'admin-respond-to-auth-challenge' command.`
    );
  }
};

const storeAccessToken = (token) => {
  process.env.AccessToken = token;
};

const adminRespondToAuthChallengeHandler = async (commands) => {
  const [, username, totp] = commands;

  try {
    verifyUsername(username);
    verifyTotp(totp);
  }
};
```



```
const [userPoolId, clientId] = getFirstEntry(FILE_USER_POOLS);
const session = process.env.SESSION;

const { AuthenticationResult } = await adminRespondToAuthChallenge({
  clientId,
  userPoolId,
  username,
  totp,
  session,
});

storeAccessToken(AuthenticationResult.AccessToken);

log("Successfully authenticated.");
} catch (err) {
  log(err);
}
};

export { adminRespondToAuthChallengeHandler };

const respondToAuthChallenge = ({
  clientId,
  username,
  session,
  userPoolId,
  code,
}) => {
  const client = new CognitoIdentityProviderClient({});

  const command = new RespondToAuthChallengeCommand({
    ChallengeName: ChallengeNameType.SOFTWARE_TOKEN_MFA,
    ChallengeResponses: {
      SOFTWARE_TOKEN_MFA_CODE: code,
      USERNAME: username,
    },
    ClientId: clientId,
    UserPoolId: userPoolId,
    Session: session,
  });

  return client.send(command);
};
```

```
import { log } from "@aws-doc-sdk-examples/lib/utils/util-log.js";
import { verifySoftwareToken } from "../../../../../actions/verify-software-token.js";

const validateTotp = (totp) => {
  if (!totp) {
    throw new Error(
      `Time-based one-time password (TOTP) must be provided to the 'validate-
software-token' command.`
    );
  }
};

const verifySoftwareTokenHandler = async (commands) => {
  const [, totp] = commands;

  try {
    validateTotp(totp);

    log("Verifying TOTP.");
    await verifySoftwareToken(totp);
    log("TOTP Verified. Run 'admin-initiate-auth' again to sign-in.");
  } catch (err) {
    console.log(err);
  }
};

export { verifySoftwareTokenHandler };

const verifySoftwareToken = (totp) => {
  const client = new CognitoIdentityProviderClient({});

  // The 'Session' is provided in the response to 'AssociateSoftwareToken'.
  const session = process.env.SESSION;

  if (!session) {
    throw new Error(
      "Missing a valid Session. Did you run 'admin-initiate-auth'?",
    );
  }

  const command = new VerifySoftwareTokenCommand({
    Session: session,
    UserCode: totp,
  });
};
```

```
    return client.send(command);  
};
```

- For API details, see the following topics in *AWS SDK for JavaScript API Reference*.
 - [AdminGetUser](#)
 - [AdminInitiateAuth](#)
 - [AdminRespondToAuthChallenge](#)
 - [AssociateSoftwareToken](#)
 - [ConfirmDevice](#)
 - [ConfirmSignUp](#)
 - [InitiateAuth](#)
 - [ListUsers](#)
 - [ResendConfirmationCode](#)
 - [RespondToAuthChallenge](#)
 - [SignUp](#)
 - [VerifySoftwareToken](#)

Amazon Comprehend examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with Amazon Comprehend.

Scenarios are code examples that show you how to accomplish specific tasks by calling multiple functions within a service or combined with other AWS services.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Topics

- [Scenarios](#)

Scenarios

Build an Amazon Transcribe streaming app

The following code example shows how to build an app that records, transcribes, and translates live audio in real-time, and emails the results.

SDK for JavaScript (v3)

Shows how to use Amazon Transcribe to build an app that records, transcribes, and translates live audio in real-time, and emails the results using Amazon Simple Email Service (Amazon SES).

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- Amazon Comprehend
- Amazon SES
- Amazon Transcribe
- Amazon Translate

Building an Amazon Lex chatbot

The following code example shows how to create a chatbot to engage your website visitors.

SDK for JavaScript (v3)

Shows how to use the Amazon Lex API to create a Chatbot within a web application to engage your web site visitors.

For complete source code and instructions on how to set up and run, see the full example [Building an Amazon Lex chatbot](#) in the AWS SDK for JavaScript developer guide.

Services used in this example

- Amazon Comprehend
- Amazon Lex
- Amazon Translate

Create an application to analyze customer feedback

The following code example shows how to create an application that analyzes customer comment cards, translates them from their original language, determines their sentiment, and generates an audio file from the translated text.

SDK for JavaScript (v3)

This example application analyzes and stores customer feedback cards. Specifically, it fulfills the need of a fictitious hotel in New York City. The hotel receives feedback from guests in various languages in the form of physical comment cards. That feedback is uploaded into the app through a web client. After an image of a comment card is uploaded, the following steps occur:

- Text is extracted from the image using Amazon Textract.
- Amazon Comprehend determines the sentiment of the extracted text and its language.
- The extracted text is translated to English using Amazon Translate.
- Amazon Polly synthesizes an audio file from the extracted text.

The full app can be deployed with the AWS CDK. For source code and deployment instructions, see the project in [GitHub](#). The following excerpts show how the AWS SDK for JavaScript is used inside of Lambda functions.

```
import {
  ComprehendClient,
  DetectDominantLanguageCommand,
  DetectSentimentCommand,
} from "@aws-sdk/client-comprehend";

/**
 * Determine the language and sentiment of the extracted text.
 *
 * @param {{ source_text: string }} extractTextOutput
 */
export const handler = async (extractTextOutput) => {
  const comprehendClient = new ComprehendClient({});

  const detectDominantLanguageCommand = new DetectDominantLanguageCommand({
    Text: extractTextOutput.source_text,
  });

  // The source language is required for sentiment analysis and
  // translation in the next step.
```

```
const { Languages } = await comprehendClient.send(
  detectDominantLanguageCommand,
);

const languageCode = Languages[0].LanguageCode;

const detectSentimentCommand = new DetectSentimentCommand({
  Text: extractTextOutput.source_text,
  LanguageCode: languageCode,
});

const { Sentiment } = await comprehendClient.send(detectSentimentCommand);

return {
  sentiment: Sentiment,
  language_code: languageCode,
};
};
```

```
import {
  DetectDocumentTextCommand,
  TextractClient,
} from "@aws-sdk/client-textract";

/**
 * Fetch the S3 object from the event and analyze it using Amazon Textract.
 *
 * @param {import("@types/aws-lambda").EventBridgeEvent<"Object Created">}
  eventBridgeS3Event
 */
export const handler = async (eventBridgeS3Event) => {
  const textractClient = new TextractClient();

  const detectDocumentTextCommand = new DetectDocumentTextCommand({
    Document: {
      S3Object: {
        Bucket: eventBridgeS3Event.bucket,
        Name: eventBridgeS3Event.object,
      },
    },
  });

  // Textract returns a list of blocks. A block can be a line, a page, word, etc.
```

```
// Each block also contains geometry of the detected text.
// For more information on the Block type, see https://docs.aws.amazon.com/
textract/latest/dg/API_Block.html.
const { Blocks } = await textractClient.send(detectDocumentTextCommand);

// For the purpose of this example, we are only interested in words.
const extractedWords = Blocks.filter((b) => b.BlockType === "WORD").map(
  (b) => b.Text,
);

return extractedWords.join(" ");
};
```

```
import { PollyClient, SynthesizeSpeechCommand } from "@aws-sdk/client-polly";
import { S3Client } from "@aws-sdk/client-s3";
import { Upload } from "@aws-sdk/lib-storage";

/**
 * Synthesize an audio file from text.
 *
 * @param {{ bucket: string, translated_text: string, object: string }}
sourceDestinationConfig
 */
export const handler = async (sourceDestinationConfig) => {
  const pollyClient = new PollyClient({});

  const synthesizeSpeechCommand = new SynthesizeSpeechCommand({
    Engine: "neural",
    Text: sourceDestinationConfig.translated_text,
    VoiceId: "Ruth",
    OutputFormat: "mp3",
  });

  const { AudioStream } = await pollyClient.send(synthesizeSpeechCommand);

  const audioKey = `${sourceDestinationConfig.object}.mp3`;

  // Store the audio file in S3.
  const s3Client = new S3Client();
  const upload = new Upload({
    client: s3Client,
    params: {
      Bucket: sourceDestinationConfig.bucket,
```

```
    Key: audioKey,  
    Body: AudioStream,  
    ContentType: "audio/mp3",  
  },  
});  
  
await upload.done();  
return audioKey;  
};
```

```
import {  
  TranslateClient,  
  TranslateTextCommand,  
} from "@aws-sdk/client-translate";  
  
/**  
 * Translate the extracted text to English.  
 *  
 * @param {{ extracted_text: string, source_language_code: string }}  
 textAndSourceLanguage  
 */  
export const handler = async (textAndSourceLanguage) => {  
  const translateClient = new TranslateClient({});  
  
  const translateCommand = new TranslateTextCommand({  
    SourceLanguageCode: textAndSourceLanguage.source_language_code,  
    TargetLanguageCode: "en",  
    Text: textAndSourceLanguage.extracted_text,  
  });  
  
  const { TranslatedText } = await translateClient.send(translateCommand);  
  
  return { translated_text: TranslatedText };  
};
```

Services used in this example

- Amazon Comprehend
- Lambda
- Amazon Polly
- Amazon Textract
- Amazon Translate

Amazon DocumentDB examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with Amazon DocumentDB.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Topics

- [Serverless examples](#)

Serverless examples

Invoke a Lambda function from a Amazon DocumentDB trigger

The following code example shows how to implement a Lambda function that receives an event triggered by receiving records from a DocumentDB change stream. The function retrieves the DocumentDB payload and logs the record contents.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [Serverless examples](#) repository.

Consuming a Amazon DocumentDB event with Lambda using JavaScript.

```
console.log('Loading function');
exports.handler = async (event, context) => {
  event.events.forEach(record => {
    logDocumentDBEvent(record);
  });
  return 'OK';
};

const logDocumentDBEvent = (record) => {
  console.log('Operation type: ' + record.event.operationType);
  console.log('db: ' + record.event.ns.db);
};
```

```
    console.log('collection: ' + record.event.ns.coll);
    console.log('Full document:', JSON.stringify(record.event.fullDocument, null,
2));
};
```

Consuming a Amazon DocumentDB event with Lambda using TypeScript

```
import { DocumentDBEventRecord, DocumentDBEventSubscriptionContext } from 'aws-
lambda';

console.log('Loading function');

export const handler = async (
  event: DocumentDBEventSubscriptionContext,
  context: any
): Promise<string> => {
  event.events.forEach((record: DocumentDBEventRecord) => {
    logDocumentDBEvent(record);
  });
  return 'OK';
};

const logDocumentDBEvent = (record: DocumentDBEventRecord): void => {
  console.log('Operation type: ' + record.event.operationType);
  console.log('db: ' + record.event.ns.db);
  console.log('collection: ' + record.event.ns.coll);
  console.log('Full document:', JSON.stringify(record.event.fullDocument, null, 2));
};
```

DynamoDB examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with DynamoDB.

Basics are code examples that show you how to perform the essential operations within a service.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Scenarios are code examples that show you how to accomplish specific tasks by calling multiple functions within a service or combined with other AWS services.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Get started

Hello DynamoDB

The following code examples show how to get started using DynamoDB.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

For more details on working with DynamoDB in AWS SDK for JavaScript, see [Programming DynamoDB with JavaScript](#).

```
import { ListTablesCommand, DynamoDBClient } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({});

export const main = async () => {
  const command = new ListTablesCommand({});

  const response = await client.send(command);
  console.log(response.TableNames.join("\n"));
  return response;
};
```

- For API details, see [ListTables](#) in *AWS SDK for JavaScript API Reference*.

Topics

- [Basics](#)
- [Actions](#)

- [Scenarios](#)
- [Serverless examples](#)

Basics

Learn the basics

The following code example shows how to:

- Create a table that can hold movie data.
- Put, get, and update a single movie in the table.
- Write movie data to the table from a sample JSON file.
- Query for movies that were released in a given year.
- Scan for movies that were released in a range of years.
- Delete a movie from the table, then delete the table.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { readFileSync } from "fs";
import {
  BillingMode,
  CreateTableCommand,
  DeleteTableCommand,
  DynamoDBClient,
  waitUntilTableExists,
} from "@aws-sdk/client-dynamodb";

/**
 * This module is a convenience library. It abstracts Amazon DynamoDB's data type
 * descriptors (such as S, N, B, and B00L) by marshalling JavaScript objects into
 * AttributeValue shapes.
 */
```

```
import {
  BatchWriteCommand,
  DeleteCommand,
  DynamoDBDocumentClient,
  GetCommand,
  PutCommand,
  UpdateCommand,
  paginateQuery,
  paginateScan,
} from "@aws-sdk/lib-dynamodb";

// These modules are local to our GitHub repository. We recommend cloning
// the project from GitHub if you want to run this example.
// For more information, see https://github.com/awsdocs/aws-doc-sdk-examples.
import { getUniqueName } from "@aws-doc-sdk-examples/lib/utils/util-string.js";
import { dirnameFromMetaUrl } from "@aws-doc-sdk-examples/lib/utils/util-fs.js";
import { chunkArray } from "@aws-doc-sdk-examples/lib/utils/util-array.js";

const dirname = dirnameFromMetaUrl(import.meta.url);
const tableName = getUniqueName("Movies");
const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

const log = (msg) => console.log(`[SCENARIO] ${msg}`);

export const main = async () => {
  /**
   * Create a table.
   */

  const createTableCommand = new CreateTableCommand({
    TableName: tableName,
    // This example performs a large write to the database.
    // Set the billing mode to PAY_PER_REQUEST to
    // avoid throttling the large write.
    BillingMode: BillingMode.PAY_PER_REQUEST,
    // Define the attributes that are necessary for the key schema.
    AttributeDefinitions: [
      {
        AttributeName: "year",
        // 'N' is a data type descriptor that represents a number type.
        // For a list of all data type descriptors, see the following link.
        // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Programming.LowLevelAPI.html#Programming.LowLevelAPI.DataTypeDescriptors

```

```
        AttributeType: "N",
    },
    { AttributeName: "title", AttributeType: "S" },
  ],
  // The KeySchema defines the primary key. The primary key can be
  // a partition key, or a combination of a partition key and a sort key.
  // Key schema design is important. For more info, see
  // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/best-
practices.html
  KeySchema: [
    // The way your data is accessed determines how you structure your keys.
    // The movies table will be queried for movies by year. It makes sense
    // to make year our partition (HASH) key.
    { AttributeName: "year", KeyType: "HASH" },
    { AttributeName: "title", KeyType: "RANGE" },
  ],
});

log("Creating a table.");
const createTableResponse = await client.send(createTableCommand);
log(`Table created: ${JSON.stringify(createTableResponse.TableDescription)}`);

// This polls with DescribeTableCommand until the requested table is 'ACTIVE'.
// You can't write to a table before it's active.
log("Waiting for the table to be active.");
await waitUntilTableExists({ client }, { TableName: tableName });
log("Table active.");

/**
 * Add a movie to the table.
 */

log("Adding a single movie to the table.");
// PutCommand is the first example usage of 'lib-dynamodb'.
const putCommand = new PutCommand({
  TableName: tableName,
  Item: {
    // In 'client-dynamodb', the AttributeValue would be required ( `year: { N:
1981 }` )
    // 'lib-dynamodb' simplifies the usage ( `year: 1981` )
    year: 1981,
    // The preceding KeySchema defines 'title' as our sort (RANGE) key, so 'title'
    // is required.
    title: "The Evil Dead",
  },
});
```

```
// Every other attribute is optional.
info: {
  genres: ["Horror"],
},
},
});
await docClient.send(putCommand);
log("The movie was added.");

/**
 * Get a movie from the table.
 */

log("Getting a single movie from the table.");
const getCommand = new GetCommand({
  TableName: tableName,
  // Requires the complete primary key. For the movies table, the primary key
  // is only the id (partition key).
  Key: {
    year: 1981,
    title: "The Evil Dead",
  },
  // Set this to make sure that recent writes are reflected.
  // For more information, see https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadConsistency.html.
  ConsistentRead: true,
});
const getResponse = await docClient.send(getCommand);
log(`Got the movie: ${JSON.stringify(getResponse.Item)}`);

/**
 * Update a movie in the table.
 */

log("Updating a single movie in the table.");
const updateCommand = new UpdateCommand({
  TableName: tableName,
  Key: { year: 1981, title: "The Evil Dead" },
  // This update expression appends "Comedy" to the list of genres.
  // For more information on update expressions, see
  // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Expressions.UpdateExpressions.html
  UpdateExpression: "set #i.#g = list_append(#i.#g, :vals)",
  ExpressionAttributeNames: { "#i": "info", "#g": "genres" },
});
```

```
    ExpressionAttributeValues: {
      ":vals": ["Comedy"],
    },
    ReturnValues: "ALL_NEW",
  });
const updateResponse = await docClient.send(updateCommand);
log(`Movie updated: ${JSON.stringify(updateResponse.Attributes)}`);

/**
 * Delete a movie from the table.
 */

log("Deleting a single movie from the table.");
const deleteCommand = new DeleteCommand({
  TableName: tableName,
  Key: { year: 1981, title: "The Evil Dead" },
});
await client.send(deleteCommand);
log("Movie deleted.");

/**
 * Upload a batch of movies.
 */

log("Adding movies from local JSON file.");
const file = readFileSync(
  `${dirname}../../../../resources/sample_files/movies.json`,
);
const movies = JSON.parse(file.toString());
// chunkArray is a local convenience function. It takes an array and returns
// a generator function. The generator function yields every N items.
const movieChunks = chunkArray(movies, 25);
// For every chunk of 25 movies, make one BatchWrite request.
for (const chunk of movieChunks) {
  const putRequests = chunk.map((movie) => ({
    PutRequest: {
      Item: movie,
    },
  }));
}

const command = new BatchWriteCommand({
  RequestItems: {
    [tableName]: putRequests,
  },
});
```



```
});

    await docClient.send(command);
}
log("Movies added.");

/**
 * Query for movies by year.
 */

log("Querying for all movies from 1981.");
const paginatedQuery = paginateQuery(
    { client: docClient },
    {
        TableName: tableName,
        //For more information about query expressions, see
        // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
        Query.html#Query.KeyConditionExpressions
        KeyConditionExpression: "#y = :y",
        // 'year' is a reserved word in DynamoDB. Indicate that it's an attribute
        // name by using an expression attribute name.
        ExpressionAttributeNames: { "#y": "year" },
        ExpressionAttributeValues: { ":y": 1981 },
        ConsistentRead: true,
    },
);
/**
 * @type { Record<string, any>[] };
 */
const movies1981 = [];
for await (const page of paginatedQuery) {
    movies1981.push(...page.Items);
}
log(`Movies: ${movies1981.map((m) => m.title).join(", ")}`);

/**
 * Scan the table for movies between 1980 and 1990.
 */

log(`Scan for movies released between 1980 and 1990`);
// A 'Scan' operation always reads every item in the table. If your design
requires
// the use of 'Scan', consider indexing your table or changing your design.
```

```

// https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-query-
scan.html
const paginatedScan = paginateScan(
  { client: docClient },
  {
    TableName: tableName,
    // Scan uses a filter expression instead of a key condition expression. Scan
will
    // read the entire table and then apply the filter.
    FilterExpression: "#y between :y1 and :y2",
    ExpressionAttributeNames: { "#y": "year" },
    ExpressionAttributeValues: { ":y1": 1980, ":y2": 1990 },
    ConsistentRead: true,
  },
);
/**
 * @type { Record<string, any>[] };
 */
const movies1980to1990 = [];
for await (const page of paginatedScan) {
  movies1980to1990.push(...page.Items);
}
log(
  `Movies: ${movies1980to1990
    .map((m) => `${m.title} (${m.year})`)
    .join(", ")}`);
);

/**
 * Delete the table.
 */

const deleteTableCommand = new DeleteTableCommand({ TableName: tableName });
log(`Deleting table ${tableName}.`);
await client.send(deleteTableCommand);
log("Table deleted.");
};

```

- For API details, see the following topics in *AWS SDK for JavaScript API Reference*.
 - [BatchWriteItem](#)
 - [CreateTable](#)

- [DeleteItem](#)
- [DeleteTable](#)
- [DescribeTable](#)
- [GetItem](#)
- [PutItem](#)
- [Query](#)
- [Scan](#)
- [UpdateItem](#)

Actions

BatchExecuteStatement

The following code example shows how to use BatchExecuteStatement.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create a batch of items using PartiQL.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";

import {
  DynamoDBDocumentClient,
  BatchExecuteStatementCommand,
} from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const breakfastFoods = ["Eggs", "Bacon", "Sausage"];
  const command = new BatchExecuteStatementCommand({
```

```
Statements: breakfastFoods.map((food) => ({
  Statement: `INSERT INTO BreakfastFoods value {'Name':?}`,
  Parameters: [food],
})),
});

const response = await docClient.send(command);
console.log(response);
return response;
};
```

Get a batch of items using PartiQL.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";

import {
  DynamoDBDocumentClient,
  BatchExecuteStatementCommand,
} from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new BatchExecuteStatementCommand({
    Statements: [
      {
        Statement: "SELECT * FROM PepperMeasurements WHERE Unit=?",
        Parameters: ["Teaspoons"],
        ConsistentRead: true,
      },
      {
        Statement: "SELECT * FROM PepperMeasurements WHERE Unit=?",
        Parameters: ["Grams"],
        ConsistentRead: true,
      },
    ],
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

```
};
```

Update a batch of items using PartiQL.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";

import {
  DynamoDBDocumentClient,
  BatchExecuteStatementCommand,
} from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const eggUpdates = [
    ["duck", "fried"],
    ["chicken", "omelette"],
  ];
  const command = new BatchExecuteStatementCommand({
    Statements: eggUpdates.map((change) => ({
      Statement: "UPDATE Eggs SET Style=? where Variety=?",
      Parameters: [change[1], change[0]],
    })),
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

Delete a batch of items using PartiQL.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";

import {
  DynamoDBDocumentClient,
  BatchExecuteStatementCommand,
} from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
```

```
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new BatchExecuteStatementCommand({
    Statements: [
      {
        Statement: "DELETE FROM Flavors where Name=?",
        Parameters: ["Grape"],
      },
      {
        Statement: "DELETE FROM Flavors where Name=?",
        Parameters: ["Strawberry"],
      },
    ],
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

- For API details, see [BatchExecuteStatement](#) in *AWS SDK for JavaScript API Reference*.

BatchGetItem

The following code example shows how to use `BatchGetItem`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

This example uses the document client to simplify working with items in DynamoDB. For API details see [BatchGet](#).

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { BatchGetCommand, DynamoDBDocumentClient } from "@aws-sdk/lib-dynamodb";
```

```
const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new BatchGetCommand({
    // Each key in this object is the name of a table. This example refers
    // to a Books table.
    RequestItems: {
      Books: {
        // Each entry in Keys is an object that specifies a primary key.
        Keys: [
          {
            Title: "How to AWS",
          },
          {
            Title: "DynamoDB for DBAs",
          },
        ],
        // Only return the "Title" and "PageCount" attributes.
        ProjectionExpression: "Title, PageCount",
      },
    },
  });

  const response = await docClient.send(command);
  console.log(response.Responses["Books"]);
  return response;
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [BatchGetItem](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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);
    });
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [BatchGetItem](#) in *AWS SDK for JavaScript API Reference*.

BatchWriteItem

The following code example shows how to use BatchWriteItem.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

This example uses the document client to simplify working with items in DynamoDB. For API details see [BatchWrite](#).

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import {
  BatchWriteCommand,
  DynamoDBDocumentClient,
} from "@aws-sdk/lib-dynamodb";
import { readFileSync } from "fs";

// These modules are local to our GitHub repository. We recommend cloning
// the project from GitHub if you want to run this example.
// For more information, see https://github.com/awsdocs/aws-doc-sdk-examples.
import { dirnameFromMetaUrl } from "@aws-doc-sdk-examples/lib/utils/util-fs.js";
import { chunkArray } from "@aws-doc-sdk-examples/lib/utils/util-array.js";

const dirname = dirnameFromMetaUrl(import.meta.url);

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const file = readFileSync(
    `${dirname}../../../../resources/sample_files/movies.json`,
  );

  const movies = JSON.parse(file.toString());

  // chunkArray is a local convenience function. It takes an array and returns
  // a generator function. The generator function yields every N items.
  const movieChunks = chunkArray(movies, 25);

  // For every chunk of 25 movies, make one BatchWrite request.
  for (const chunk of movieChunks) {
```

```
const putRequests = chunk.map((movie) => ({
  PutRequest: {
    Item: movie,
  },
}));

const command = new BatchWriteCommand({
  RequestItems: {
    // An existing table is required. A composite key of 'title' and 'year' is
    recommended
    // to account for duplicate titles.
    ["BatchWriteMoviesTable"]: putRequests,
  },
});

await docClient.send(command);
}
};
```

- For API details, see [BatchWriteItem](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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);
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [BatchWriteItem](#) in *AWS SDK for JavaScript API Reference*.

CreateTable

The following code example shows how to use CreateTable.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { CreateTableCommand, DynamoDBClient } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({});

export const main = async () => {
  const command = new CreateTableCommand({
    TableName: "EspressoDrinks",
    // For more information about data types,
    // see https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
    // HowItWorks.NamingRulesDataTypes.html#HowItWorks.DataTypes and
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
    // Programming.LowLevelAPI.html#Programming.LowLevelAPI.DataTypeDescriptors
    AttributeDefinitions: [
      {
        AttributeName: "DrinkName",
        AttributeType: "S",
      },
    ],
    KeySchema: [
      {
        AttributeName: "DrinkName",
        KeyType: "HASH",
      },
    ],
    ProvisionedThroughput: {
      ReadCapacityUnits: 1,
      WriteCapacityUnits: 1,
    },
  });

  const response = await client.send(command);
  console.log(response);
  return response;
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [CreateTable](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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);
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [CreateTable](#) in *AWS SDK for JavaScript API Reference*.

DeleteItem

The following code example shows how to use DeleteItem.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

This example uses the document client to simplify working with items in DynamoDB. For API details see [DeleteCommand](#).

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, DeleteCommand } from "@aws-sdk/lib-dynamodb";
```

```
const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new DeleteCommand({
    TableName: "Sodas",
    Key: {
      Flavor: "Cola",
    },
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [DeleteItem](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Delete an item from a table.

```
// 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);
    }
  });
});
```

Delete an item from a table using the DynamoDB document client.

```
// 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);
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [DeleteItem](#) in *AWS SDK for JavaScript API Reference*.

DeleteTable

The following code example shows how to use DeleteTable.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { DeleteTableCommand, DynamoDBClient } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({});

export const main = async () => {
  const command = new DeleteTableCommand({
    TableName: "DecafCoffees",
  });

  const response = await client.send(command);
  console.log(response);
  return response;
};
```

- For API details, see [DeleteTable](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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);
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [DeleteTable](#) in *AWS SDK for JavaScript API Reference*.

DescribeTable

The following code example shows how to use DescribeTable.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { DescribeTableCommand, DynamoDBClient } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({});

export const main = async () => {
  const command = new DescribeTableCommand({
    TableName: "Pastries",
```

```
});

const response = await client.send(command);
console.log(`TABLE NAME: ${response.Table.TableName}`);
console.log(`TABLE ITEM COUNT: ${response.Table.ItemCount}`);
return response;
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [DescribeTable](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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);
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [DescribeTable](#) in *AWS SDK for JavaScript API Reference*.

DescribeTimeToLive

The following code example shows how to use `DescribeTimeToLive`.

SDK for JavaScript (v3)

```
import { DynamoDBClient, DescribeTimeToLiveCommand } from "@aws-sdk/client-dynamodb";

const describeTableTTL = async (tableName, region) => {
  const client = new DynamoDBClient({
    region: region,
    endpoint: `https://dynamodb.${region}.amazonaws.com`
  });

  try {
    const ttlDescription = await client.send(new
DescribeTimeToLiveCommand({ TableName: tableName }));

    if (ttlDescription.TimeToLiveDescription.TimeToLiveStatus === 'ENABLED') {
      console.log("TTL is enabled for table %s.", tableName);
    } else {
      console.log("TTL is not enabled for table %s.", tableName);
    }

    return ttlDescription;
  } catch (e) {
    console.error(`Error describing table: ${e}`);
    throw e;
  }
}

// enter table name and change region if desired.
describeTableTTL('your-table-name', 'us-east-1');
```

- For API details, see [DescribeTimeToLive](#) in *AWS SDK for JavaScript API Reference*.

ExecuteStatement

The following code example shows how to use ExecuteStatement.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create an item using PartiQL.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";

import {
  ExecuteStatementCommand,
  DynamoDBDocumentClient,
} from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new ExecuteStatementCommand({
    Statement: `INSERT INTO Flowers value {'Name':?}`,
    Parameters: ["Rose"],
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

Get an item using PartiQL.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";

import {
  ExecuteStatementCommand,
```

```
DynamoDBDocumentClient,  
} from "@aws-sdk/lib-dynamodb";  
  
const client = new DynamoDBClient({});  
const docClient = DynamoDBDocumentClient.from(client);  
  
export const main = async () => {  
  const command = new ExecuteStatementCommand({  
    Statement: "SELECT * FROM CloudTypes WHERE IsStorm=?",  
    Parameters: [false],  
    ConsistentRead: true,  
  });  
  
  const response = await docClient.send(command);  
  console.log(response);  
  return response;  
};
```

Update an item using PartiQL.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";  
  
import {  
  ExecuteStatementCommand,  
  DynamoDBDocumentClient,  
} from "@aws-sdk/lib-dynamodb";  
  
const client = new DynamoDBClient({});  
const docClient = DynamoDBDocumentClient.from(client);  
  
export const main = async () => {  
  const command = new ExecuteStatementCommand({  
    Statement: "UPDATE EyeColors SET IsRecessive=? where Color=?",  
    Parameters: [true, "blue"],  
  });  
  
  const response = await docClient.send(command);  
  console.log(response);  
  return response;  
};
```

Delete an item using PartiQL.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";

import {
  ExecuteStatementCommand,
  DynamoDBDocumentClient,
} from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new ExecuteStatementCommand({
    Statement: "DELETE FROM PaintColors where Name=?",
    Parameters: ["Purple"],
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

- For API details, see [ExecuteStatement](#) in *AWS SDK for JavaScript API Reference*.

GetItem

The following code example shows how to use `GetItem`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

This example uses the document client to simplify working with items in DynamoDB. For API details see [GetCommand](#).

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, GetCommand } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new GetCommand({
    TableName: "AngryAnimals",
    Key: {
      CommonName: "Shoebill",
    },
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

- For API details, see [GetItem](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Get an item from a table.

```
// 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);
  }
});
```

Get an item from a table using the DynamoDB document client.

```
// 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);
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [GetItem](#) in *AWS SDK for JavaScript API Reference*.

ListTables

The following code example shows how to use `ListTables`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { ListTablesCommand, DynamoDBClient } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({});

export const main = async () => {
  const command = new ListTablesCommand({});

  const response = await client.send(command);
  console.log(response);
  return response;
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [ListTables](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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);
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [ListTables](#) in *AWS SDK for JavaScript API Reference*.

PutItem

The following code example shows how to use PutItem.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

This example uses the document client to simplify working with items in DynamoDB. For API details see [PutCommand](#).

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { PutCommand, DynamoDBDocumentClient } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new PutCommand({
    TableName: "HappyAnimals",
    Item: {
```

```
        CommonName: "Shiba Inu",
    },
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
```

- For API details, see [PutItem](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Put an item in a table.

```
// 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);
  }
});
```

Put an item in a table using the DynamoDB document client.

```
// 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);
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [PutItem](#) in *AWS SDK for JavaScript API Reference*.

Query

The following code example shows how to use Query.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

This example uses the document client to simplify working with items in DynamoDB. For API details see [QueryCommand](#).

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { QueryCommand, DynamoDBDocumentClient } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new QueryCommand({
    TableName: "CoffeeCrop",
    KeyConditionExpression:
      "OriginCountry = :originCountry AND RoastDate > :roastDate",
    ExpressionAttributeValues: {
      ":originCountry": "Ethiopia",
      ":roastDate": "2023-05-01",
    },
  },
  ConsistentRead: true,
});

const response = await docClient.send(command);
console.log(response);
return response;
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [Query](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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);
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [Query](#) in *AWS SDK for JavaScript API Reference*.

Scan

The following code example shows how to use Scan.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

This example uses the document client to simplify working with items in DynamoDB. For API details see [ScanCommand](#).

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, ScanCommand } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

export const main = async () => {
  const command = new ScanCommand({
    ProjectionExpression: "#Name, Color, AvgLifeSpan",
    ExpressionAttributeNames: { "#Name": "Name" },
    TableName: "Birds",
  });

  const response = await docClient.send(command);
  for (const bird of response.Items) {
    console.log(`${bird.Name} - (${bird.Color}, ${bird.AvgLifeSpan})`);
  }
  return response;
};
```

- For API details, see [Scan](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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 + ")"
      );
    });
  }
});
```

```
    }  
  });
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [Scan](#) in *AWS SDK for JavaScript API Reference*.

UpdateItem

The following code example shows how to use `UpdateItem`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

This example uses the document client to simplify working with items in DynamoDB. For API details see [UpdateCommand](#).

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";  
import { DynamoDBDocumentClient, UpdateCommand } from "@aws-sdk/lib-dynamodb";  
  
const client = new DynamoDBClient({});  
const docClient = DynamoDBDocumentClient.from(client);  
  
export const main = async () => {  
  const command = new UpdateCommand({  
    TableName: "Dogs",  
    Key: {  
      Breed: "Labrador",  
    },  
    UpdateExpression: "set Color = :color",  
    ExpressionAttributeValues: {  
      ":color": "black",  
    },  
    ReturnValues: "ALL_NEW",  
  });
```

```
const response = await docClient.send(command);
console.log(response);
return response;
};
```

- For API details, see [UpdateItem](#) in *AWS SDK for JavaScript API Reference*.

UpdateTimeToLive

The following code example shows how to use UpdateTimeToLive.

SDK for JavaScript (v3)

Enable TTL on an existing DynamoDB table.

```
import { DynamoDBClient, UpdateTimeToLiveCommand } from "@aws-sdk/client-dynamodb";

const enableTTL = async (tableName, ttlAttribute) => {

  const client = new DynamoDBClient({});

  const params = {
    TableName: tableName,
    TimeToLiveSpecification: {
      Enabled: true,
      AttributeName: ttlAttribute
    }
  };

  try {
    const response = await client.send(new UpdateTimeToLiveCommand(params));
    if (response.$metadata.httpStatusCode === 200) {
      console.log(`TTL enabled successfully for table ${tableName}, using
attribute name ${ttlAttribute}.`);
    } else {
      console.log(`Failed to enable TTL for table ${tableName}, response
object: ${response}`);
    }
    return response;
  } catch (e) {
    console.error(`Error enabling TTL: ${e}`);
  }
};
```

```
        throw e;
    }
};

// call with your own values
enableTTL('ExampleTable', 'exampleTtlAttribute');
```

Disable TTL on an existing DynamoDB table.

```
import { DynamoDBClient, UpdateTimeToLiveCommand } from "@aws-sdk/client-dynamodb";

const disableTTL = async (tableName, ttlAttribute) => {

    const client = new DynamoDBClient({});

    const params = {
        TableName: tableName,
        TimeToLiveSpecification: {
            Enabled: false,
            AttributeName: ttlAttribute
        }
    };

    try {
        const response = await client.send(new UpdateTimeToLiveCommand(params));
        if (response.$metadata.httpStatusCode === 200) {
            console.log(`TTL disabled successfully for table ${tableName}, using
attribute name ${ttlAttribute}.`);
        } else {
            console.log(`Failed to disable TTL for table ${tableName}, response
object: ${response}`);
        }
        return response;
    } catch (e) {
        console.error(`Error disabling TTL: ${e}`);
        throw e;
    }
};

// call with your own values
disableTTL('ExampleTable', 'exampleTtlAttribute');
```

- For API details, see [UpdateTimeToLive](#) in *AWS SDK for JavaScript API Reference*.

Scenarios

Build an app to submit data to a DynamoDB table

The following code example shows how to build an application that submits data to an Amazon DynamoDB table and notifies you when a user updates the table.

SDK for JavaScript (v3)

This example shows how to build an app that enables users to submit data to an Amazon DynamoDB table, and send a text message to the administrator using Amazon Simple Notification Service (Amazon SNS).

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

This example is also available in the [AWS SDK for JavaScript v3 developer guide](#).

Services used in this example

- DynamoDB
- Amazon SNS

Conditionally update an item's TTL

The following code example shows how to conditionally update an item's TTL.

SDK for JavaScript (v3)

Update TTL on on an existing DynamoDB Item in a table, with a condition.

```
import { DynamoDBClient, UpdateItemCommand } from "@aws-sdk/client-dynamodb";
import { marshall, unmarshall } from "@aws-sdk/util-dynamodb";

const updateDynamoDBItem = async (tableName, region, partitionKey, sortKey,
  newAttribute) => {
  const client = new DynamoDBClient({
    region: region,
    endpoint: `https://dynamodb.${region}.amazonaws.com`
  });
```

```
const currentTime = Math.floor(Date.now() / 1000);

const params = {
  TableName: tableName,
  Key: marshall({
    artist: partitionKey,
    album: sortKey
  }),
  UpdateExpression: "SET newAttribute = :newAttribute",
  ConditionExpression: "expireAt > :expiration",
  ExpressionAttributeValues: marshall({
    ':newAttribute': newAttribute,
    ':expiration': currentTime
  }),
  ReturnValues: "ALL_NEW"
};

try {
  const response = await client.send(new UpdateItemCommand(params));
  const responseData = unmarshall(response.Attributes);
  console.log("Item updated successfully: ", responseData);
  return responseData;
} catch (error) {
  if (error.name === "ConditionalCheckFailedException") {
    console.log("Condition check failed: Item's 'expireAt' is expired.");
  } else {
    console.error("Error updating item: ", error);
  }
  throw error;
}
};

// Enter your values here
updateDynamoDBItem('your-table-name', "us-east-1", 'your-partition-key-value', 'your-sort-key-value', 'your-new-attribute-value');
```

- For API details, see [UpdateItem](#) in *AWS SDK for JavaScript API Reference*.

Create a serverless application to manage photos

The following code example shows how to create a serverless application that lets users manage photos using labels.

SDK for JavaScript (v3)

Shows how to develop a photo asset management application that detects labels in images using Amazon Rekognition and stores them for later retrieval.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

For a deep dive into the origin of this example see the post on [AWS Community](#).

Services used in this example

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Create a web application to track DynamoDB data

The following code example shows how to create a web application that tracks work items in an Amazon DynamoDB table and uses Amazon Simple Email Service (Amazon SES) to send reports.

SDK for JavaScript (v3)

Shows how to use the Amazon DynamoDB API to create a dynamic web application that tracks DynamoDB work data.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- DynamoDB
- Amazon SES

Create an item with a TTL

The following code example shows how to create an item with TTL.

SDK for JavaScript (v3)

```
import { DynamoDBClient, PutItemCommand } from "@aws-sdk/client-dynamodb";

function createDynamoDBItem(table_name, region, partition_key, sort_key) {
  const client = new DynamoDBClient({
    region: region,
    endpoint: `https://dynamodb.${region}.amazonaws.com`
  });

  // Get the current time in epoch second format
  const current_time = Math.floor(new Date().getTime() / 1000);

  // Calculate the expireAt time (90 days from now) in epoch second format
  const expire_at = Math.floor((new Date().getTime() + 90 * 24 * 60 * 60 * 1000) /
1000);

  // Create DynamoDB item
  const item = {
    'partitionKey': {'S': partition_key},
    'sortKey': {'S': sort_key},
    'createdAt': {'N': current_time.toString()},
    'expireAt': {'N': expire_at.toString()}
  };

  const putItemCommand = new PutItemCommand({
    TableName: table_name,
    Item: item,
    ProvisionedThroughput: {
      ReadCapacityUnits: 1,
      WriteCapacityUnits: 1,
    },
  });

  client.send(putItemCommand, function(err, data) {
    if (err) {
      console.log("Exception encountered when creating item %s, here's what
happened: ", data, ex);
      throw err;
    } else {
      console.log("Item created successfully: %s.", data);
      return data;
    }
  });
}
```



```
}  
  
// use your own values  
createDynamoDBItem('your-table-name', 'us-east-1', 'your-partition-key-value',  
  'your-sort-key-value');
```

- For API details, see [PutItem](#) in *AWS SDK for JavaScript API Reference*.

Detect PPE in images

The following code example shows how to build an app that uses Amazon Rekognition to detect Personal Protective Equipment (PPE) in images.

SDK for JavaScript (v3)

Shows how to use Amazon Rekognition with the AWS SDK for JavaScript to create an application to detect personal protective equipment (PPE) in images located in an Amazon Simple Storage Service (Amazon S3) bucket. The app saves the results to an Amazon DynamoDB table, and sends the admin an email notification with the results using Amazon Simple Email Service (Amazon SES).

Learn how to:

- Create an unauthenticated user using Amazon Cognito.
- Analyze images for PPE using Amazon Rekognition.
- Verify an email address for Amazon SES.
- Update a DynamoDB table with results.
- Send an email notification using Amazon SES.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- DynamoDB
- Amazon Rekognition
- Amazon S3
- Amazon SES

Invoke a Lambda function from a browser

The following code example shows how to invoke an AWS Lambda function from a browser.

SDK for JavaScript (v2)

You can create a browser-based application that uses an AWS Lambda function to update an Amazon DynamoDB table with user selections.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- DynamoDB
- Lambda

SDK for JavaScript (v3)

You can create a browser-based application that uses an AWS Lambda function to update an Amazon DynamoDB table with user selections. This app uses AWS SDK for JavaScript v3.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- DynamoDB
- Lambda

Query a table by using batches of PartiQL statements

The following code example shows how to:

- Get a batch of items by running multiple SELECT statements.
- Add a batch of items by running multiple INSERT statements.
- Update a batch of items by running multiple UPDATE statements.
- Delete a batch of items by running multiple DELETE statements.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Execute batch PartiQL statements.

```
import {
  BillingMode,
  CreateTableCommand,
  DeleteTableCommand,
  DescribeTableCommand,
  DynamoDBClient,
  waitUntilTableExists,
} from "@aws-sdk/client-dynamodb";
import {
  DynamoDBDocumentClient,
  BatchExecuteStatementCommand,
} from "@aws-sdk/lib-dynamodb";
import { ScenarioInput } from "@aws-doc-sdk-examples/lib/scenario";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

const log = (msg) => console.log(`[SCENARIO] ${msg}`);
const tableName = "Cities";

export const main = async (confirmAll = false) => {
  /**
   * Delete table if it exists.
   */
  try {
    await client.send(new DescribeTableCommand({ TableName: tableName }));
    // If no error was thrown, the table exists.
    const input = new ScenarioInput(
      "deleteTable",
      `A table named ${tableName} already exists. If you choose not to delete
this table, the scenario cannot continue. Delete it?`,
      { confirmAll },
    );
```

```
const deleteTable = await input.handle({});
if (deleteTable) {
  await client.send(new DeleteTableCommand({ tableName }));
} else {
  console.warn(
    "Scenario could not run. Either delete ${tableName} or provide a unique
table name.",
  );
  return;
}
} catch (caught) {
  if (
    caught instanceof Error &&
    caught.name === "ResourceNotFoundException"
  ) {
    // Do nothing. This means the table is not there.
  } else {
    throw caught;
  }
}

/**
 * Create a table.
 */

log("Creating a table.");
const createTableCommand = new CreateTableCommand({
  TableName: tableName,
  // This example performs a large write to the database.
  // Set the billing mode to PAY_PER_REQUEST to
  // avoid throttling the large write.
  BillingMode: BillingMode.PAY_PER_REQUEST,
  // Define the attributes that are necessary for the key schema.
  AttributeDefinitions: [
    {
      AttributeName: "name",
      // 'S' is a data type descriptor that represents a number type.
      // For a list of all data type descriptors, see the following link.
      // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
Programming.LowLevelAPI.html#Programming.LowLevelAPI.DataTypeDescriptors
      AttributeType: "S",
    },
  ],
});
// The KeySchema defines the primary key. The primary key can be
```

```
// a partition key, or a combination of a partition key and a sort key.
// Key schema design is important. For more info, see
// https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/best-
practices.html
    KeySchema: [{ AttributeName: "name", KeyType: "HASH" }],
  });
await client.send(createTableCommand);
log(`Table created: ${tableName}.`);

/**
 * Wait until the table is active.
 */

// This polls with DescribeTableCommand until the requested table is 'ACTIVE'.
// You can't write to a table before it's active.
log("Waiting for the table to be active.");
await waitUntilTableExists({ client }, { TableName: tableName });
log("Table active.");

/**
 * Insert items.
 */

log("Inserting cities into the table.");
const addItemStatementCommand = new BatchExecuteStatementCommand({
  // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ql-
reference.insert.html
  Statements: [
    {
      Statement: `INSERT INTO ${tableName} value {'name':?, 'population':?}`,
      Parameters: ["Alachua", 10712],
    },
    {
      Statement: `INSERT INTO ${tableName} value {'name':?, 'population':?}`,
      Parameters: ["High Springs", 6415],
    },
  ],
});
await docClient.send(addItemsStatementCommand);
log(`Cities inserted.`);

/**
 * Select items.
 */
```

```
log("Selecting cities from the table.");
const selectItemsStatementCommand = new BatchExecuteStatementCommand({
  // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ql-
reference.select.html
  Statements: [
    {
      Statement: `SELECT * FROM ${tableName} WHERE name=?`,
      Parameters: ["Alachua"],
    },
    {
      Statement: `SELECT * FROM ${tableName} WHERE name=?`,
      Parameters: ["High Springs"],
    },
  ],
});
const selectItemResponse = await docClient.send(selectItemsStatementCommand);
log(
  `Got cities: ${selectItemResponse.Responses.map(
    (r) => `${r.Item.name} (${r.Item.population})`,
  )}.join(", ")`,
);

/**
 * Update items.
 */

log("Modifying the populations.");
const updateItemStatementCommand = new BatchExecuteStatementCommand({
  // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ql-
reference.update.html
  Statements: [
    {
      Statement: `UPDATE ${tableName} SET population=? WHERE name=?`,
      Parameters: [10, "Alachua"],
    },
    {
      Statement: `UPDATE ${tableName} SET population=? WHERE name=?`,
      Parameters: [5, "High Springs"],
    },
  ],
});
await docClient.send(updateItemStatementCommand);
log(`Updated cities.`);
```

```
/**
 * Delete the items.
 */

log("Deleting the cities.");
const deleteItemStatementCommand = new BatchExecuteStatementCommand({
  // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ql-
reference.delete.html
  Statements: [
    {
      Statement: `DELETE FROM ${tableName} WHERE name=?`,
      Parameters: ["Alachua"],
    },
    {
      Statement: `DELETE FROM ${tableName} WHERE name=?`,
      Parameters: ["High Springs"],
    },
  ],
});
await docClient.send(deleteItemStatementCommand);
log("Cities deleted.");

/**
 * Delete the table.
 */

log("Deleting the table.");
const deleteTableCommand = new DeleteTableCommand({ TableName: tableName });
await client.send(deleteTableCommand);
log("Table deleted.");
};
```

- For API details, see [BatchExecuteStatement](#) in *AWS SDK for JavaScript API Reference*.

Query a table using PartiQL

The following code example shows how to:

- Get an item by running a SELECT statement.
- Add an item by running an INSERT statement.

- Update an item by running an UPDATE statement.
- Delete an item by running a DELETE statement.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Execute single PartiQL statements.

```
import {
  BillingMode,
  CreateTableCommand,
  DeleteTableCommand,
  DescribeTableCommand,
  DynamoDBClient,
  waitUntilTableExists,
} from "@aws-sdk/client-dynamodb";
import {
  DynamoDBDocumentClient,
  ExecuteStatementCommand,
} from "@aws-sdk/lib-dynamodb";
import { ScenarioInput } from "@aws-doc-sdk-examples/lib/scenario";

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

const log = (msg) => console.log(`[SCENARIO] ${msg}`);
const tableName = "SingleOriginCoffees";

export const main = async (confirmAll = false) => {
  /**
   * Delete table if it exists.
   */
  try {
    await client.send(new DescribeTableCommand({ TableName: tableName }));
    // If no error was thrown, the table exists.
    const input = new ScenarioInput(
```



```
    "deleteTable",
    `A table named ${tableName} already exists. If you choose not to delete
this table, the scenario cannot continue. Delete it?`,
    { confirmAll },
  );
const deleteTable = await input.handle({});
if (deleteTable) {
  await client.send(new DeleteTableCommand({ tableName }));
} else {
  console.warn(
    "Scenario could not run. Either delete ${tableName} or provide a unique
table name.",
  );
  return;
}
} catch (caught) {
  if (
    caught instanceof Error &&
    caught.name === "ResourceNotFoundException"
  ) {
    // Do nothing. This means the table is not there.
  } else {
    throw caught;
  }
}

/**
 * Create a table.
 */

log("Creating a table.");
const createTableCommand = new CreateTableCommand({
  TableName: tableName,
  // This example performs a large write to the database.
  // Set the billing mode to PAY_PER_REQUEST to
  // avoid throttling the large write.
  BillingMode: BillingMode.PAY_PER_REQUEST,
  // Define the attributes that are necessary for the key schema.
  AttributeDefinitions: [
    {
      AttributeName: "varietal",
      // 'S' is a data type descriptor that represents a number type.
      // For a list of all data type descriptors, see the following link.

```

```
    // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
    Programming.LowLevelAPI.html#Programming.LowLevelAPI.DataTypeDescriptors
    AttributeType: "S",
  },
],
// The KeySchema defines the primary key. The primary key can be
// a partition key, or a combination of a partition key and a sort key.
// Key schema design is important. For more info, see
// https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/best-
practices.html
KeySchema: [{ AttributeName: "varietal", KeyType: "HASH" }],
});
await client.send(createTableCommand);
log(`Table created: ${tableName}.`);

/**
 * Wait until the table is active.
 */

// This polls with DescribeTableCommand until the requested table is 'ACTIVE'.
// You can't write to a table before it's active.
log("Waiting for the table to be active.");
await waitUntilTableExists({ client }, { TableName: tableName });
log("Table active.");

/**
 * Insert an item.
 */

log("Inserting a coffee into the table.");
const addItemStatementCommand = new ExecuteStatementCommand({
  // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ql-
reference.insert.html
  Statement: `INSERT INTO ${tableName} value {'varietal':?, 'profile':?}`,
  Parameters: ["arabica", ["chocolate", "floral"]],
});
await client.send(addItemStatementCommand);
log(`Coffee inserted.`);

/**
 * Select an item.
 */

log("Selecting the coffee from the table.");
```

```
const selectItemStatementCommand = new ExecuteStatementCommand({
  // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ql-
reference.select.html
  Statement: `SELECT * FROM ${tableName} WHERE varietal=?`,
  Parameters: ["arabica"],
});
const selectItemResponse = await docClient.send(selectItemStatementCommand);
log(`Got coffee: ${JSON.stringify(selectItemResponse.Items[0])}`);

/**
 * Update the item.
 */

log("Add a flavor profile to the coffee.");
const updateItemStatementCommand = new ExecuteStatementCommand({
  // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ql-
reference.update.html
  Statement: `UPDATE ${tableName} SET profile=list_append(profile, ?) WHERE
varietal=?`,
  Parameters: [["fruity"], "arabica"],
});
await client.send(updateItemStatementCommand);
log(`Updated coffee`);

/**
 * Delete the item.
 */

log("Deleting the coffee.");
const deleteItemStatementCommand = new ExecuteStatementCommand({
  // https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ql-
reference.delete.html
  Statement: `DELETE FROM ${tableName} WHERE varietal=?`,
  Parameters: ["arabica"],
});
await docClient.send(deleteItemStatementCommand);
log("Coffee deleted.");

/**
 * Delete the table.
 */

log("Deleting the table.");
const deleteTableCommand = new DeleteTableCommand({ TableName: tableName });
```

```
await client.send(deleteTableCommand);
log("Table deleted.");
};
```

- For API details, see [ExecuteStatement](#) in *AWS SDK for JavaScript API Reference*.

Query for TTL items

The following code example shows how to query for TTL items.

SDK for JavaScript (v3)

```
import { DynamoDBClient, QueryCommand } from "@aws-sdk/client-dynamodb";
import { marshall, unmarshall } from "@aws-sdk/util-dynamodb";

async function queryDynamoDBItems(tableName, region, primaryKey) {
  const client = new DynamoDBClient({
    region: region,
    endpoint: `https://dynamodb.${region}.amazonaws.com`
  });

  const currentTime = Math.floor(Date.now() / 1000);

  const params = {
    TableName: tableName,
    KeyConditionExpression: "#pk = :pk",
    FilterExpression: "#ea > :ea",
    ExpressionAttributeNames: {
      "#pk": "primaryKey",
      "#ea": "expireAt"
    },
    ExpressionAttributeValues: marshall({
      ":pk": primaryKey,
      ":ea": currentTime
    })
  };

  try {
    const { Items } = await client.send(new QueryCommand(params));
    Items.forEach(item => {
      console.log(unmarshall(item))
    });
  }
};
```

```
        return Items;
    } catch (err) {
        console.error(`Error querying items: ${err}`);
        throw err;
    }
}

//enter your own values here
queryDynamoDBItems('your-table-name', 'your-partition-key-value');
```

- For API details, see [Query](#) in *AWS SDK for JavaScript API Reference*.

Update an item's TTL

The following code example shows how to update an item's TTL.

SDK for JavaScript (v3)

```
import { DynamoDBClient, UpdateItemCommand } from "@aws-sdk/client-dynamodb";
import { marshall, unmarshall } from "@aws-sdk/util-dynamodb";

async function updateDynamoDBItem(tableName, region, partitionKey, sortKey) {
    const client = new DynamoDBClient({
        region: region,
        endpoint: `https://dynamodb.${region}.amazonaws.com`
    });

    const currentTime = Math.floor(Date.now() / 1000);
    const expireAt = Math.floor((Date.now() + 90 * 24 * 60 * 60 * 1000) / 1000);

    const params = {
        TableName: tableName,
        Key: marshall({
            partitionKey: partitionKey,
            sortKey: sortKey
        }),
        UpdateExpression: "SET updatedAt = :c, expireAt = :e",
        ExpressionAttributeValues: marshall({
            ":c": currentTime,
            ":e": expireAt
        }),
    };
};
```

```
try {
  const data = await client.send(new UpdateItemCommand(params));
  const responseData = unmarshall(data.Attributes);
  console.log("Item updated successfully: %s", responseData);
  return responseData;
} catch (err) {
  console.error("Error updating item:", err);
  throw err;
}

//enter your values here
updateDynamoDBItem('your-table-name', 'us-east-1', 'your-partition-key-value',
  'your-sort-key-value');
```

- For API details, see [UpdateItem](#) in *AWS SDK for JavaScript API Reference*.

Serverless examples

Invoke a Lambda function from a DynamoDB trigger

The following code example shows how to implement a Lambda function that receives an event triggered by receiving records from a DynamoDB stream. The function retrieves the DynamoDB payload and logs the record contents.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [Serverless examples](#) repository.

Consuming a DynamoDB event with Lambda using JavaScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
```

```
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(record => {
      logDynamoDBRecord(record);
    });
  };

const logDynamoDBRecord = (record) => {
  console.log(record.eventID);
  console.log(record.eventName);
  console.log(`DynamoDB Record: ${JSON.stringify(record.dynamodb)}`);
};
```

Consuming a DynamoDB event with Lambda using TypeScript.

```
export const handler = async (event, context) => {
  console.log(JSON.stringify(event, null, 2));
  event.Records.forEach(record => {
    logDynamoDBRecord(record);
  });
}

const logDynamoDBRecord = (record) => {
  console.log(record.eventID);
  console.log(record.eventName);
  console.log(`DynamoDB Record: ${JSON.stringify(record.dynamodb)}`);
};
```

Reporting batch item failures for Lambda functions with a DynamoDB trigger

The following code example shows how to implement partial batch response for Lambda functions that receive events from a DynamoDB stream. The function reports the batch item failures in the response, signaling to Lambda to retry those messages later.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [Serverless examples](#) repository.

Reporting DynamoDB batch item failures with Lambda using JavaScript.

```
export const handler = async (event) => {
  const records = event.Records;
  let curRecordSequenceNumber = "";

  for (const record of records) {
    try {
      // Process your record
      curRecordSequenceNumber = record.dynamodb.SequenceNumber;
    } catch (e) {
      // Return failed record's sequence number
      return { batchItemFailures: [{ itemIdentifier: curRecordSequenceNumber }] };
    }
  }

  return { batchItemFailures: [] };
};
```

Reporting DynamoDB batch item failures with Lambda using TypeScript.

```
import {
  DynamoDBBatchResponse,
  DynamoDBBatchItemFailure,
  DynamoDBStreamEvent,
} from "aws-lambda";

export const handler = async (
  event: DynamoDBStreamEvent
): Promise<DynamoDBBatchResponse> => {
  const batchItemFailures: DynamoDBBatchItemFailure[] = [];
  let curRecordSequenceNumber;

  for (const record of event.Records) {
    curRecordSequenceNumber = record.dynamodb?.SequenceNumber;

    if (curRecordSequenceNumber) {
      batchItemFailures.push({
        itemIdentifier: curRecordSequenceNumber,
      });
    }
  }
}
```



```
    return { batchItemFailures: batchItemFailures };  
};
```

Amazon EC2 examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with Amazon EC2.

Basics are code examples that show you how to perform the essential operations within a service.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Scenarios are code examples that show you how to accomplish specific tasks by calling multiple functions within a service or combined with other AWS services.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Get started

Hello Amazon EC2

The following code examples show how to get started using Amazon EC2.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { DescribeSecurityGroupsCommand, EC2Client } from "@aws-sdk/client-ec2";  
  
// Call DescribeSecurityGroups and display the result.  
export const main = async () => {  
    const client = new EC2Client();
```

```
try {
  const { SecurityGroups } = await client.send(
    new DescribeSecurityGroupsCommand({}),
  );

  const securityGroupList = SecurityGroups.slice(0, 9)
    .map((sg) => ` • ${sg.GroupId}: ${sg.GroupName}`)
    .join("\n");

  console.log(
    "Hello, Amazon EC2! Let's list up to 10 of your security groups:",
  );
  console.log(securityGroupList);
} catch (err) {
  console.error(err);
}
};

// Call function if run directly.
import { fileURLToPath } from "url";
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  main();
}
```

- For API details, see [DescribeSecurityGroups](#) in *AWS SDK for JavaScript API Reference*.

Topics

- [Basics](#)
- [Actions](#)
- [Scenarios](#)

Basics

Learn the basics

The following code example shows how to:

- Create a key pair and security group.
- Select an Amazon Machine Image (AMI) and compatible instance type, then create an instance.

- Stop and restart the instance.
- Associate an Elastic IP address with your instance.
- Connect to your instance with SSH, then clean up resources.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

This file contains a list of common actions used with EC2. The steps are constructed with a Scenario framework that simplifies running an interactive example. For the full context, visit the GitHub repository.

```
import { tmpdir } from "node:os";
import { writeFile, mkdtemp, rm } from "node:fs/promises";
import { join } from "node:path";
import { get } from "node:http";

import {
  AllocateAddressCommand,
  AssociateAddressCommand,
  AuthorizeSecurityGroupIngressCommand,
  CreateKeyPairCommand,
  CreateSecurityGroupCommand,
  DeleteKeyPairCommand,
  DeleteSecurityGroupCommand,
  DisassociateAddressCommand,
  paginateDescribeImages,
  paginateDescribeInstances,
  paginateDescribeInstanceTypes,
  ReleaseAddressCommand,
  RunInstancesCommand,
  StartInstancesCommand,
  StopInstancesCommand,
  TerminateInstancesCommand,
  waitUntilInstanceStatusOk,
  waitUntilInstanceStopped,
  waitUntilInstanceTerminated,
```

```
} from "@aws-sdk/client-ec2";

import {
  ScenarioAction,
  ScenarioInput,
  ScenarioOutput,
} from "@aws-doc-sdk-examples/lib/scenario/index.js";

import { paginateGetParametersByPath, SSMClient } from "@aws-sdk/client-ssm";

/**
 * @typedef {{
 *   ec2Client: import('@aws-sdk/client-ec2').EC2Client,
 *   errors: Error[],
 *   keyPairId?: string,
 *   tmpDirectory?: string,
 *   securityGroupId?: string,
 *   ipAddress?: string,
 *   images?: import('@aws-sdk/client-ec2').Image[],
 *   image?: import('@aws-sdk/client-ec2').Image,
 *   instanceTypes?: import('@aws-sdk/client-ec2').InstanceTypeInfo[],
 *   instanceId?: string,
 *   instanceIpAddress?: string,
 *   allocationId?: string,
 *   allocatedIpAddress?: string,
 *   associationId?: string,
 * }} State
 */

/**
 * A skip function provided to the `skipWhen` of a Step when you want
 * to ignore that step if any errors have occurred.
 * @param {State} state
 */
const skipWhenErrors = (state) => state.errors.length > 0;

const MAX_WAITER_TIME_IN_SECONDS = 60 * 8;

export const confirm = new ScenarioInput("confirmContinue", "Continue?", {
  type: "confirm",
  skipWhen: skipWhenErrors,
});

export const exitOnNoConfirm = new ScenarioAction(
```

```
`exitOnConfirmContinueFalse`,
(** @type { { earlyExit: boolean } & Record<string, any> } */ state) => {
  if (!state[confirm.name]) {
    state.earlyExit = true;
  }
},
{
  skipWhen: skipWhenErrors,
},
);

export const greeting = new ScenarioOutput(
  "greeting",
  `Welcome to the Amazon EC2 basic usage scenario.
Before you launch an instances, you'll need to provide a few things:


- A key pair - This is for SSH access to your EC2 instance. You only need to provide the name.
- A security group - This is used for configuring access to your instance. Again, only the name is needed.
- An IP address - Your public IP address will be fetched.
- An Amazon Machine Image (AMI)
- A compatible instance type`,
  { header: true, preformatted: true, skipWhen: skipWhenErrors },
);

export const provideKeyPairName = new ScenarioInput(
  "keyPairName",
  "Provide a name for a new key pair.",
  { type: "input", default: "ec2-example-key-pair", skipWhen: skipWhenErrors },
);

export const createKeyPair = new ScenarioAction(
  "createKeyPair",
  async (** @type {State} */ state) => {
    try {
      // Create a key pair in Amazon EC2.
      const { KeyMaterial, KeyPairId } = await state.ec2Client.send(
        // A unique name for the key pair. Up to 255 ASCII characters.
        new CreateKeyPairCommand({ KeyName: state[provideKeyPairName.name] }),
      );

      state.keyPairId = KeyPairId;

      // Save the private key in a temporary location.

```

```
state.tmpDirectory = await mkdtemp(join(tmpdir(), "ec2-scenario-tmp"));
await writeFile(
  `${state.tmpDirectory}/${state[provideKeyPairName.name]}.pem`,
  KeyMaterial,
  {
    mode: 0o400,
  },
);
} catch (caught) {
  if (
    caught instanceof Error &&
    caught.name === "InvalidKeyPair.Duplicate"
  ) {
    caught.message = `${caught.message}. Try another key name.`;
  }

  state.errors.push(caught);
}
},
{ skipWhen: skipWhenErrors },
);

export const logKeyPair = new ScenarioOutput(
  "logKeyPair",
  (/** @type {State} */ state) =>
    `Created the key pair ${state[provideKeyPairName.name]}.\`,
  { skipWhen: skipWhenErrors },
);

export const confirmDeleteKeyPair = new ScenarioInput(
  "confirmDeleteKeyPair",
  "Do you want to delete the key pair?",
  {
    type: "confirm",
    // Don't do anything when a key pair was never created.
    skipWhen: (/** @type {State} */ state) => !state.keyPairId,
  },
);

export const maybeDeleteKeyPair = new ScenarioAction(
  "deleteKeyPair",
  async (/** @type {State} */ state) => {
    try {
      // Delete a key pair by name from EC2
```

```
    await state.ec2Client.send(
      new DeleteKeyPairCommand({ KeyName: state[provideKeyPairName.name] }),
    );
  } catch (caught) {
    if (
      caught instanceof Error &&
      // Occurs when a required parameter (e.g. KeyName) is undefined.
      caught.name === "MissingParameter"
    ) {
      caught.message = `${caught.message}. Did you provide the required value?`;
    }
    state.errors.push(caught);
  }
},
{
  // Don't do anything when there's no key pair to delete or the user chooses
  // to keep it.
  skipWhen: (/** @type {State} */ state) =>
    !state.keyPairId || !state[confirmDeleteKeyPair.name],
},
);

export const provideSecurityGroupName = new ScenarioInput(
  "securityGroupName",
  "Provide a name for a new security group.",
  { type: "input", default: "ec2-scenario-sg", skipWhen: skipWhenErrors },
);

export const createSecurityGroup = new ScenarioAction(
  "createSecurityGroup",
  async (/** @type {State} */ state) => {
    try {
      // Create a new security group that will be used to configure ingress/egress
      for
      // an EC2 instance.
      const { GroupId } = await state.ec2Client.send(
        new CreateSecurityGroupCommand({
          GroupName: state[provideSecurityGroupName.name],
          Description: "A security group for the Amazon EC2 example.",
        }),
      );
      state.securityGroupId = GroupId;
    } catch (caught) {
      if (caught instanceof Error && caught.name === "InvalidGroup.Duplicate") {
```

```
        caught.message = `${caught.message}. Please provide a different name for
your security group.`;
    }

    state.errors.push(caught);
  }
},
{ skipWhen: skipWhenErrors },
);

export const logSecurityGroup = new ScenarioOutput(
  "logSecurityGroup",
  (/** @type {State} */ state) =>
    `Created the security group ${state.securityGroupId}.`,
  { skipWhen: skipWhenErrors },
);

export const confirmDeleteSecurityGroup = new ScenarioInput(
  "confirmDeleteSecurityGroup",
  "Do you want to delete the security group?",
  {
    type: "confirm",
    // Don't do anything when a security group was never created.
    skipWhen: (/** @type {State} */ state) => !state.securityGroupId,
  },
);

export const maybeDeleteSecurityGroup = new ScenarioAction(
  "deleteSecurityGroup",
  async (/** @type {State} */ state) => {
    try {
      // Delete the security group if the 'skipWhen' condition below is not met.
      await state.ec2Client.send(
        new DeleteSecurityGroupCommand({
          GroupId: state.securityGroupId,
        }),
      );
    } catch (caught) {
      if (
        caught instanceof Error &&
        caught.name === "InvalidGroupId.Malformed"
      ) {
        caught.message = `${caught.message}. Please provide a valid GroupId.`;
      }
    }
  }
);
```



```

    state.errors.push(caught);
  }
},
{
  // Don't do anything when there's no security group to delete
  // or the user chooses to keep it.
  skipWhen: (/** @type {State} */ state) =>
    !state.securityGroupId || !state[confirmDeleteSecurityGroup.name],
},
);

export const authorizeSecurityGroupIngress = new ScenarioAction(
  "authorizeSecurity",
  async (/** @type {State} */ state) => {
    try {
      // Get the public IP address of the machine running this example.
      const ipAddress = await new Promise((res, rej) => {
        get("http://checkip.amazonaws.com", (response) => {
          let data = "";
          response.on("data", (chunk) => (data += chunk));
          response.on("end", () => res(data.trim()));
        }).on("error", (err) => {
          rej(err);
        });
      });
      state[`ipAddress`] = ipAddress;
      // Allow ingress from the IP address above to the security group.
      // This will allow you to SSH into the EC2 instance.
      const command = new AuthorizeSecurityGroupIngressCommand({
        GroupId: state.securityGroupId,
        IpPermissions: [
          {
            IpProtocol: "tcp",
            FromPort: 22,
            ToPort: 22,
            IpRanges: [{ CidrIp: `${ipAddress}/32` }],
          },
        ],
      });

      await state.ec2Client.send(command);
    } catch (caught) {
      if (
        caught instanceof Error &&

```

```
        caught.name === "InvalidGroupId.Malformed"
    ) {
        caught.message = `${caught.message}. Please provide a valid GroupId.`;
    }

    state.errors.push(caught);
}
},
{ skipWhen: skipWhenErrors },
);

export const logSecurityGroupIngress = new ScenarioOutput(
    "logSecurityGroupIngress",
    (** @type {State} */ state) =>
    `Allowed SSH access from your public IP: ${state.ipAddress}.`,
    { skipWhen: skipWhenErrors },
);

export const getImages = new ScenarioAction(
    "images",
    async (** @type {State} */ state) => {
        const AMIs = [];
        // Some AWS services publish information about common artifacts as AWS Systems
        // Manager (SSM)
        // public parameters. For example, the Amazon Elastic Compute Cloud (Amazon EC2)
        // service publishes information about Amazon Machine Images (AMIs) as public
        // parameters.

        // Create the paginator for getting images. Actions that return multiple pages
        // of
        // results have paginators to simplify those calls.
        const getParametersByPathPaginator = paginateGetParametersByPath(
            {
                // Not storing this client in state since it's only used once.
                client: new SSMClient({}),
            },
            {
                // The path to the public list of the latest amazon-linux instances.
                Path: "/aws/service/ami-amazon-linux-latest",
            },
        );
    });

    try {
        for await (const page of getParametersByPathPaginator) {
```

```
    page.Parameters.forEach((param) => {
      // Filter by Amazon Linux 2
      if (param.Name.includes("amzn2")) {
        AMIs.push(param.Value);
      }
    });
  }
} catch (caught) {
  if (caught instanceof Error && caught.name === "InvalidFilterValue") {
    caught.message = `${caught.message} Please provide a valid filter value for
paginateGetParametersByPath.`;
  }
  state.errors.push(caught);
  return;
}

const imageDetails = [];
const describeImagesPaginator = paginateDescribeImages(
  { client: state.ec2Client },
  // The images found from the call to SSM.
  { ImageIds: AMIs },
);

try {
  // Get more details for the images found above.
  for await (const page of describeImagesPaginator) {
    imageDetails.push(...(page.Images || []));
  }

  // Store the image details for later use.
  state["images"] = imageDetails;
} catch (caught) {
  if (caught instanceof Error && caught.name === "InvalidAMIID.NotFound") {
    caught.message = `${caught.message}. Please provide a valid image id.`;
  }

  state.errors.push(caught);
}
},
{ skipWhen: skipWhenErrors },
);

export const provideImage = new ScenarioInput(
  "image",
```

```
"Select one of the following images.",
{
  type: "select",
  choices: (/** @type { State } */ state) =>
    state.images.map((image) => ({
      name: `${image.ImageId} - ${image.Description}`,
      value: image,
    })),
  default: (/** @type { State } */ state) => state.images[0],
  skipWhen: skipWhenErrors,
},
);

export const getCompatibleInstanceTypes = new ScenarioAction(
  "getCompatibleInstanceTypes",
  async (/** @type {State} */ state) => {
    // Get more details about instance types that match the architecture of
    // the provided image.
    const paginator = paginateDescribeInstanceTypes(
      { client: state.ec2Client, pageSize: 25 },
      {
        Filters: [
          {
            Name: "processor-info.supported-architecture",
            // The value selected from provideImage()
            Values: [state.image.Architecture],
          },
          // Filter for smaller, less expensive, types.
          { Name: "instance-type", Values: ["*.micro", "*.small"] },
        ],
      },
    );

    const instanceTypes = [];

    try {
      for await (const page of paginator) {
        if (page.InstanceTypes.length) {
          instanceTypes.push...(page.InstanceTypes || []);
        }
      }

      if (!instanceTypes.length) {
        state.errors.push(
```

```
        "No instance types matched the instance type filters.",
    );
  }
} catch (caught) {
  if (caught instanceof Error && caught.name === "InvalidParameterValue") {
    caught.message = `${caught.message}. Please check the provided values and
try again.`;
  }

  state.errors.push(caught);
}

state["instanceTypes"] = instanceTypes;
},
{ skipWhen: skipWhenErrors },
);

export const provideInstanceType = new ScenarioInput(
  "instanceType",
  "Select an instance type.",
  {
    choices: (/** @type {State} */ state) =>
      state.instanceTypes.map((instanceType) => ({
        name: `${instanceType.InstanceType} - Memory:
${instanceType.MemoryInfo.SizeInMiB}`,
        value: instanceType.InstanceType,
      })),
    type: "select",
    default: (/** @type {State} */ state) =>
      state.instanceTypes[0].InstanceType,
    skipWhen: skipWhenErrors,
  },
);

export const runInstance = new ScenarioAction(
  "runInstance",
  async (/** @type { State } */ state) => {
    const { Instances } = await state.ec2Client.send(
      new RunInstancesCommand({
        KeyName: state[provideKeyPairName.name],
        SecurityGroupIds: [state.securityGroupId],
        ImageId: state.image.ImageId,
        InstanceType: state[provideInstanceType.name],
      })
    );
  }
);
```

```
    // Availability Zones have capacity limitations that may impact your ability
    // to launch instances.
    // The `RunInstances` operation will only succeed if it can allocate at
    // least the `MinCount` of instances.
    // However, EC2 will attempt to launch up to the `MaxCount` of instances,
    // even if the full request cannot be satisfied.
    // If you need a specific number of instances, use `MinCount` and `MaxCount`
    // set to the same value.
    // If you want to launch up to a certain number of instances, use `MaxCount`
    // and let EC2 provision as many as possible.
    // If you require a minimum number of instances, but do not want to exceed a
    // maximum, use both `MinCount` and `MaxCount`.
    MinCount: 1,
    MaxCount: 1,
  })),
);

state.instanceId = Instances[0].InstanceId;

try {
  // Poll `DescribeInstanceStatus` until status is "ok".
  await waitUntilInstanceStatusOk(
    {
      client: state.ec2Client,
      maxWaitTime: MAX_WAITER_TIME_IN_SECONDS,
    },
    { InstanceIds: [Instances[0].InstanceId] },
  );
} catch (caught) {
  if (caught instanceof Error && caught.name === "TimeoutError") {
    caught.message = `${caught.message}. Try increasing the maxWaitTime in the
waiter.`;
  }

  state.errors.push(caught);
}
},
{ skipWhen: skipWhenErrors },
);

export const logRunInstance = new ScenarioOutput(
  "logRunInstance",
  "The next step is to run your EC2 instance for the first time. This can take a few
minutes.",
```

```
    { header: true, skipWhen: skipWhenErrors },
  );

export const describeInstance = new ScenarioAction(
  "describeInstance",
  async (/** @type { State } */ state) => {
    /** @type { import("@aws-sdk/client-ec2").Instance[] } */
    const instances = [];

    try {
      const paginator = paginateDescribeInstances(
        {
          client: state.ec2Client,
        },
        {
          // Only get our created instance.
          InstanceIds: [state.instanceId],
        },
      );
    }

    for await (const page of paginator) {
      for (const reservation of page.Reservations) {
        instances.push(...reservation.Instances);
      }
    }
    if (instances.length !== 1) {
      throw new Error(`Instance ${state.instanceId} not found.`);
    }

    // The only info we need is the IP address for SSH purposes.
    state.instanceIpAddress = instances[0].PublicIpAddress;
  } catch (caught) {
    if (caught instanceof Error && caught.name === "InvalidParameterValue") {
      caught.message = `${caught.message}. Please check provided values and try
again.`;
    }

    state.errors.push(caught);
  }
},
  { skipWhen: skipWhenErrors },
);

export const logSSHConnectionInfo = new ScenarioOutput(
```

```
"logSSHConnectionInfo",
  (** @type { State } */ state) =>
  `You can now SSH into your instance using the following command:
ssh -i ${state.tmpDirectory}/${state[provideKeyPairName].pem} ec2-user@
${state.instanceIpAddress}`,
  { preformatted: true, skipWhen: skipWhenErrors },
);

export const logStopInstance = new ScenarioOutput(
  "logStopInstance",
  "Stopping your EC2 instance.",
  { skipWhen: skipWhenErrors },
);

export const stopInstance = new ScenarioAction(
  "stopInstance",
  async (** @type { State } */ state) => {
    try {
      await state.ec2Client.send(
        new StopInstancesCommand({
          InstanceIds: [state.instanceId],
        }),
      );

      await waitUntilInstanceStopped(
        {
          client: state.ec2Client,
          maxWaitTime: MAX_WAITER_TIME_IN_SECONDS,
        },
        { InstanceIds: [state.instanceId] },
      );
    } catch (caught) {
      if (caught instanceof Error && caught.name === "TimeoutError") {
        caught.message = `${caught.message}. Try increasing the maxWaitTime in the
waiter.`;
      }

      state.errors.push(caught);
    }
  },
  // Don't try to stop an instance that doesn't exist.
  { skipWhen: (** @type { State } */ state) => !state.instanceId },
);
```



```
export const logIpAddressBehavior = new ScenarioOutput(
  "logIpAddressBehavior",
  [
    "When you run an instance, by default it's assigned an IP address.",
    "That IP address is not static. It will change every time the instance is
restarted.",
    "The next step is to stop and restart your instance to demonstrate this
behavior.",
  ].join(" "),
  { header: true, skipWhen: skipWhenErrors },
);

export const logStartInstance = new ScenarioOutput(
  "logStartInstance",
  (/** @type { State } */ state) => `Starting instance ${state.instanceId}`,
  { skipWhen: skipWhenErrors },
);

export const startInstance = new ScenarioAction(
  "startInstance",
  async (/** @type { State } */ state) => {
    try {
      await state.ec2Client.send(
        new StartInstancesCommand({
          InstanceIds: [state.instanceId],
        }),
      );

      await waitUntilInstanceStatusOk(
        {
          client: state.ec2Client,
          maxWaitTime: MAX_WAITER_TIME_IN_SECONDS,
        },
        { InstanceIds: [state.instanceId] },
      );
    } catch (caught) {
      if (caught instanceof Error && caught.name === "TimeoutError") {
        caught.message = `${caught.message}. Try increasing the maxWaitTime in the
waiter.`;
      }

      state.errors.push(caught);
    }
  },
);
```

```
    { skipWhen: skipWhenErrors },
  );

export const logIpAllocation = new ScenarioOutput(
  "logIpAllocation",
  [
    "It is possible to have a static IP address.",
    "To demonstrate this, an IP will be allocated and associated to your EC2 instance.",
  ].join(" "),
  { header: true, skipWhen: skipWhenErrors },
);

export const allocateIp = new ScenarioAction(
  "allocateIp",
  async (/** @type { State } */ state) => {
    try {
      // An Elastic IP address is allocated to your AWS account, and is yours until you release it.
      const { AllocationId, PublicIp } = await state.ec2Client.send(
        new AllocateAddressCommand({}),
      );
      state.allocationId = AllocationId;
      state.allocatedIpAddress = PublicIp;
    } catch (caught) {
      if (caught instanceof Error && caught.name === "MissingParameter") {
        caught.message = `${caught.message}. Did you provide these values?`;
      }
      state.errors.push(caught);
    }
  },
  { skipWhen: skipWhenErrors },
);

export const associateIp = new ScenarioAction(
  "associateIp",
  async (/** @type { State } */ state) => {
    try {
      // Associate an allocated IP address to an EC2 instance. An IP address can be allocated
      // with the AllocateAddress action.
      const { AssociationId } = await state.ec2Client.send(
        new AssociateAddressCommand({
          AllocationId: state.allocationId,
```

```
        InstanceId: state.instanceId,
      )),
    );
    state.associationId = AssociationId;
    // Update the IP address that is being tracked to match
    // the one just associated.
    state.instanceIpAddress = state.allocatedIpAddress;
  } catch (caught) {
    if (
      caught instanceof Error &&
      caught.name === "InvalidAllocationID.NotFound"
    ) {
      caught.message = `${caught.message}. Did you provide the ID of a valid
Elastic IP address AllocationId?`;
    }
    state.errors.push(caught);
  }
},
{ skipWhen: skipWhenErrors },
);

export const logStaticIpProof = new ScenarioOutput(
  "logStaticIpProof",
  "The IP address should remain the same even after stopping and starting the
instance.",
  { header: true, skipWhen: skipWhenErrors },
);

export const logCleanUp = new ScenarioOutput(
  "logCleanUp",
  "That's it! You can choose to clean up the resources now, or clean them up on your
own later.",
  { header: true, skipWhen: skipWhenErrors },
);

export const confirmDisassociateAddress = new ScenarioInput(
  "confirmDisassociateAddress",
  "Do you want to disassociate and release the static IP address created earlier?",
  {
    type: "confirm",
    skipWhen: (/** @type { State } */ state) => !state.associationId,
  },
);
```

```
export const maybeDisassociateAddress = new ScenarioAction(
  "maybeDisassociateAddress",
  async (/** @type { State } */ state) => {
    try {
      await state.ec2Client.send(
        new DisassociateAddressCommand({
          AssociationId: state.associationId,
        }),
      );
    } catch (caught) {
      if (
        caught instanceof Error &&
        caught.name === "InvalidAssociationID.NotFound"
      ) {
        caught.message = `${caught.message}. Please provide a valid association
ID.`;
      }
      state.errors.push(caught);
    }
  },
  {
    skipWhen: (/** @type { State } */ state) =>
      !state[confirmDisassociateAddress.name] || !state.associationId,
  },
);

export const maybeReleaseAddress = new ScenarioAction(
  "maybeReleaseAddress",
  async (/** @type { State } */ state) => {
    try {
      await state.ec2Client.send(
        new ReleaseAddressCommand({
          AllocationId: state.allocationId,
        }),
      );
    } catch (caught) {
      if (
        caught instanceof Error &&
        caught.name === "InvalidAllocationID.NotFound"
      ) {
        caught.message = `${caught.message}. Please provide a valid AllocationID.`;
      }
      state.errors.push(caught);
    }
  }
);
```

```

    },
    {
      skipWhen: (/** @type { State } */ state) =>
        !state[confirmDisassociateAddress.name] || !state.allocationId,
    },
  );

export const confirmTerminateInstance = new ScenarioInput(
  "confirmTerminateInstance",
  "Do you want to terminate the instance?",
  // Don't do anything when an instance was never run.
  {
    skipWhen: (/** @type { State } */ state) => !state.instanceId,
    type: "confirm",
  },
);

export const maybeTerminateInstance = new ScenarioAction(
  "terminateInstance",
  async (/** @type { State } */ state) => {
    try {
      await state.ec2Client.send(
        new TerminateInstancesCommand({
          InstanceIds: [state.instanceId],
        }),
      );
      await waitUntilInstanceTerminated(
        { client: state.ec2Client },
        { InstanceIds: [state.instanceId] },
      );
    } catch (caught) {
      if (caught instanceof Error && caught.name === "TimeoutError") {
        caught.message = `${caught.message}. Try increasing the maxWaitTime in the
waiter.`;
      }

      state.errors.push(caught);
    }
  },
  {
    // Don't do anything when there's no instance to terminate or the
    // user chooses not to terminate.
    skipWhen: (/** @type { State } */ state) =>
      !state.instanceId || !state[confirmTerminateInstance.name],
  },
);

```

```
    },
  );

export const deleteTemporaryDirectory = new ScenarioAction(
  "deleteTemporaryDirectory",
  async (/** @type { State } */ state) => {
    try {
      await rm(state.tmpDirectory, { recursive: true });
    } catch (caught) {
      state.errors.push(caught);
    }
  },
);

export const logErrors = new ScenarioOutput(
  "logErrors",
  (/** @type {State}*/ state) => {
    const errorList = state.errors
      .map((err) => `• ${err.name}: ${err.message}`)
      .join("\n");
    return `Scenario errors found:\n${errorList}`;
  },
  {
    preformatted: true,
    header: true,
    // Don't log errors when there aren't any!
    skipWhen: (/** @type {State} */ state) => state.errors.length === 0,
  },
);
```

- For API details, see the following topics in *AWS SDK for JavaScript API Reference*.
 - [AllocateAddress](#)
 - [AssociateAddress](#)
 - [AuthorizeSecurityGroupIngress](#)
 - [CreateKeyPair](#)
 - [CreateSecurityGroup](#)
 - [DeleteKeyPair](#)
 - [DeleteSecurityGroup](#)
 - [DescribeImages](#)

- [DescribeInstanceTypes](#)
- [DescribeInstances](#)
- [DescribeKeyPairs](#)
- [DescribeSecurityGroups](#)
- [DisassociateAddress](#)
- [ReleaseAddress](#)
- [RunInstances](#)
- [StartInstances](#)
- [StopInstances](#)
- [TerminateInstances](#)
- [UnmonitorInstances](#)

Actions

AllocateAddress

The following code example shows how to use `AllocateAddress`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { AllocateAddressCommand, EC2Client } from "@aws-sdk/client-ec2";

/**
 * Allocates an Elastic IP address to your AWS account.
 */
export const main = async () => {
  const client = new EC2Client({});
  const command = new AllocateAddressCommand({});

  try {
```

```

const { AllocationId, PublicIp } = await client.send(command);
console.log("A new IP address has been allocated to your account:");
console.log(`ID: ${AllocationId} Public IP: ${PublicIp}`);
console.log(
  "You can view your IP addresses in the AWS Management Console for Amazon EC2.
  Look under Network & Security > Elastic IPs",
);
} catch (caught) {
  if (caught instanceof Error && caught.name === "MissingParameter") {
    console.warn(`${caught.message}. Did you provide these values?`);
  } else {
    throw caught;
  }
}
};
import { fileURLToPath } from "url";
// Call function if run directly.
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  main();
}

```

- For API details, see [AllocateAddress](#) in *AWS SDK for JavaScript API Reference*.

AssociateAddress

The following code example shows how to use AssociateAddress.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```

import { AssociateAddressCommand, EC2Client } from "@aws-sdk/client-ec2";

/**
 * Associates an Elastic IP address, or carrier IP address (for instances that are
 * in subnets in Wavelength Zones)
 * with an instance or a network interface.

```



```
* @param {{ instanceId: string, allocationId: string }} options
*/
export const main = async ({ instanceId, allocationId }) => {
  const client = new EC2Client({});
  const command = new AssociateAddressCommand({
    // You need to allocate an Elastic IP address before associating it with an
    instance.
    // You can do that with the AllocateAddressCommand.
    AllocationId: allocationId,
    // You need to create an EC2 instance before an IP address can be associated
    with it.
    // You can do that with the RunInstancesCommand.
    InstanceId: instanceId,
  });

  try {
    const { AssociationId } = await client.send(command);
    console.log(
      `Address with allocation ID ${allocationId} is now associated with instance
${instanceId}.`,
      `The association ID is ${AssociationId}.`,
    );
  } catch (caught) {
    if (
      caught instanceof Error &&
      caught.name === "InvalidAllocationID.NotFound"
    ) {
      console.warn(
        `${caught.message}. Did you provide the ID of a valid Elastic IP address
AllocationId?`,
      );
    } else {
      throw caught;
    }
  }
};
```

- For API details, see [AssociateAddress](#) in *AWS SDK for JavaScript API Reference*.

AuthorizeSecurityGroupIngress

The following code example shows how to use `AuthorizeSecurityGroupIngress`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import {
  AuthorizeSecurityGroupIngressCommand,
  EC2Client,
} from "@aws-sdk/client-ec2";

/**
 * Adds the specified inbound (ingress) rules to a security group.
 * @param {{ groupId: string, ipAddress: string }} options
 */
export const main = async ({ groupId, ipAddress }) => {
  const client = new EC2Client({});
  const command = new AuthorizeSecurityGroupIngressCommand({
    // Use a group ID from the AWS console or
    // the DescribeSecurityGroupsCommand.
    GroupId: groupId,
    IpPermissions: [
      {
        IpProtocol: "tcp",
        FromPort: 22,
        ToPort: 22,
        // The IP address to authorize.
        // For more information on this notation, see
        // https://en.wikipedia.org/wiki/Classless_Inter-
        Domain_Routing#CIDR_notation
        IpRanges: [{ CidrIp: `${ipAddress}/32` }],
      },
    ],
  });

  try {
    const { SecurityGroupRules } = await client.send(command);
    console.log(JSON.stringify(SecurityGroupRules, null, 2));
  } catch (caught) {
    if (caught instanceof Error && caught.name === "InvalidGroupId.Malformed") {
```

```
        console.warn(`${caught.message}. Please provide a valid GroupId.`);
    } else {
        throw caught;
    }
}
};
```

- For API details, see [AuthorizeSecurityGroupIngress](#) in *AWS SDK for JavaScript API Reference*.

CreateKeyPair

The following code example shows how to use CreateKeyPair.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { CreateKeyPairCommand, EC2Client } from "@aws-sdk/client-ec2";

/**
 * Creates an ED25519 or 2048-bit RSA key pair with the specified name and in the
 * specified PEM or PPK format.
 * Amazon EC2 stores the public key and displays the private key for you to save to
 * a file.
 * @param {{ keyName: string }} options
 */
export const main = async ({ keyName }) => {
    const client = new EC2Client({});
    const command = new CreateKeyPairCommand({
        KeyName: keyName,
    });

    try {
        const { KeyMaterial, KeyName } = await client.send(command);
        console.log(KeyName);
        console.log(KeyMaterial);
    } catch (caught) {
```

```
    if (caught instanceof Error && caught.name === "InvalidKeyPair.Duplicate") {
      console.warn(`${caught.message}. Try another key name.`);
    } else {
      throw caught;
    }
  }
};
```

- For API details, see [CreateKeyPair](#) in *AWS SDK for JavaScript API Reference*.

CreateLaunchTemplate

The following code example shows how to use `CreateLaunchTemplate`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const ssmClient = new SSMClient({});
const { Parameter } = await ssmClient.send(
  new GetParameterCommand({
    Name: "/aws/service/ami-amazon-linux-latest/amzn2-ami-hvm-x86_64-gp2",
  }),
);
const ec2Client = new EC2Client({});
await ec2Client.send(
  new CreateLaunchTemplateCommand({
    LaunchTemplateName: NAMES.launchTemplateName,
    LaunchTemplateData: {
      InstanceType: "t3.micro",
      ImageId: Parameter.Value,
      IamInstanceProfile: { Name: NAMES.instanceProfileName },
      UserData: readFileSync(
        join(RESOURCES_PATH, "server_startup_script.sh"),
      ).toString("base64"),
      KeyName: NAMES.keyPairName,
    },
  }),
);
```

```
 )),
```

- For API details, see [CreateLaunchTemplate](#) in *AWS SDK for JavaScript API Reference*.

CreateSecurityGroup

The following code example shows how to use CreateSecurityGroup.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { CreateSecurityGroupCommand, EC2Client } from "@aws-sdk/client-ec2";

/**
 * Creates a security group.
 * @param {{ groupName: string, description: string }} options
 */
export const main = async ({ groupName, description }) => {
  const client = new EC2Client({});
  const command = new CreateSecurityGroupCommand({
    // Up to 255 characters in length. Cannot start with sg-.
    GroupName: groupName,
    // Up to 255 characters in length.
    Description: description,
  });

  try {
    const { GroupId } = await client.send(command);
    console.log(GroupId);
  } catch (caught) {
    if (caught instanceof Error && caught.name === "InvalidParameterValue") {
      console.warn(`${caught.message}`);
    } else {
      throw caught;
    }
  }
}
```

```
};
```

- For API details, see [CreateSecurityGroup](#) in *AWS SDK for JavaScript API Reference*.

DeleteKeyPair

The following code example shows how to use DeleteKeyPair.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { DeleteKeyPairCommand, EC2Client } from "@aws-sdk/client-ec2";

/**
 * Deletes the specified key pair, by removing the public key from Amazon EC2.
 * @param {{ keyName: string }} options
 */
export const main = async ({ keyName }) => {
  const client = new EC2Client({});
  const command = new DeleteKeyPairCommand({
    KeyName: keyName,
  });

  try {
    await client.send(command);
    console.log("Successfully deleted key pair.");
  } catch (caught) {
    if (caught instanceof Error && caught.name === "MissingParameter") {
      console.warn(`${caught.message}. Did you provide the required value?`);
    } else {
      throw caught;
    }
  }
};
```

- For API details, see [DeleteKeyPair](#) in *AWS SDK for JavaScript API Reference*.

DeleteLaunchTemplate

The following code example shows how to use DeleteLaunchTemplate.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
await client.send(  
  new DeleteLaunchTemplateCommand({  
    LaunchTemplateName: NAMES.launchTemplateName,  
  }),  
);
```

- For API details, see [DeleteLaunchTemplate](#) in *AWS SDK for JavaScript API Reference*.

DeleteSecurityGroup

The following code example shows how to use DeleteSecurityGroup.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { DeleteSecurityGroupCommand, EC2Client } from "@aws-sdk/client-ec2";  
  
/**  
 * Deletes a security group. */
```

```

* @param {{ groupId: string }} options
*/
export const main = async ({ groupId }) => {
  const client = new EC2Client({});
  const command = new DeleteSecurityGroupCommand({
    GroupId: groupId,
  });

  try {
    await client.send(command);
    console.log("Security group deleted successfully.");
  } catch (caught) {
    if (caught instanceof Error && caught.name === "InvalidGroupId.Malformed") {
      console.warn(`${caught.message}. Please provide a valid GroupId.`);
    } else {
      throw caught;
    }
  }
};

```

- For API details, see [DeleteSecurityGroup](#) in *AWS SDK for JavaScript API Reference*.

DescribeAddresses

The following code example shows how to use DescribeAddresses.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```

import { DescribeAddressesCommand, EC2Client } from "@aws-sdk/client-ec2";

/**
 * Describes the specified Elastic IP addresses or all of your Elastic IP addresses.
 * @param {{ allocationId: string }} options
 */
export const main = async ({ allocationId }) => {

```



```
const client = new EC2Client({});
const command = new DescribeAddressesCommand({
  // You can omit this property to show all addresses.
  AllocationIds: [allocationId],
});

try {
  const { Addresses } = await client.send(command);
  const addressList = Addresses.map((address) => ` • ${address.PublicIp}`);
  console.log("Elastic IP addresses:");
  console.log(addressList.join("\n"));
} catch (caught) {
  if (
    caught instanceof Error &&
    caught.name === "InvalidAllocationID.NotFound"
  ) {
    console.warn(`${caught.message}. Please provide a valid AllocationId.`);
  } else {
    throw caught;
  }
}
};
```

- For API details, see [DescribeAddresses](#) in *AWS SDK for JavaScript API Reference*.

DescribeIamInstanceProfileAssociations

The following code example shows how to use `DescribeIamInstanceProfileAssociations`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const ec2Client = new EC2Client({});
const { IamInstanceProfileAssociations } = await ec2Client.send(
  new DescribeIamInstanceProfileAssociationsCommand({
    Filters: [
```

```
        { Name: "instance-id", Values: [state.targetInstance.InstanceId] },
      ],
    })),
  );
```

- For API details, see [DescribeInstanceProfileAssociations](#) in *AWS SDK for JavaScript API Reference*.

DescribeImages

The following code example shows how to use DescribeImages.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { EC2Client, paginateDescribeImages } from "@aws-sdk/client-ec2";

/**
 * Describes the specified images (AMIs, AKIs, and ARIs) available to you or all of
 * the images available to you.
 * @param {{ architecture: string, pageSize: number }} options
 */
export const main = async ({ architecture, pageSize }) => {
  pageSize = parseInt(pageSize);
  const client = new EC2Client({});

  // The paginate function is a wrapper around the base command.
  const paginator = paginateDescribeImages(
    // Without limiting the page size, this call can take a long time. pageSize is
    just sugar for
    // the MaxResults property in the base command.
    { client, pageSize },
    {
      // There are almost 70,000 images available. Be specific with your filtering
      // to increase efficiency.
    }
  );
```

```
// See https://docs.aws.amazon.com/AWSJavaScriptSDK/v3/latest/clients/client-ec2/interfaces/describeimagescommandinput.html#filters
    Filters: [{ Name: "architecture", Values: [architecture] }],
  },
);

/**
 * @type {import('@aws-sdk/client-ec2').Image[]}
 */
const images = [];
let recordsScanned = 0;

try {
  for await (const page of paginator) {
    recordsScanned += pageSize;
    if (page.Images.length) {
      images.push(...page.Images);
      break;
    } else {
      console.log(
        `No matching image found yet. Searched ${recordsScanned} records.`
      );
    }
  }

  if (images.length) {
    console.log(
      `Found ${images.length} images:\n\n${images.map((image) =>
image.Name).join("\n")}\n`,
    );
  } else {
    console.log(
      `No matching images found. Searched ${recordsScanned} records.\n`,
    );
  }

  return images;
} catch (caught) {
  if (caught instanceof Error && caught.name === "InvalidParameterValue") {
    console.warn(`${caught.message}`);
    return [];
  }
  throw caught;
}
```

```
};
```

- For API details, see [DescribeImages](#) in *AWS SDK for JavaScript API Reference*.

DescribeInstanceTypes

The following code example shows how to use `DescribeInstanceTypes`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { EC2Client, paginateDescribeInstanceTypes } from "@aws-sdk/client-ec2";

/**
 * Describes the specified instance types. By default, all instance types for the
 * current Region are described. Alternatively, you can filter the results.
 * @param {{ pageSize: string, supportedArch: string[], freeTier: boolean }} options
 */
export const main = async ({ pageSize, supportedArch, freeTier }) => {
  pageSize = parseInt(pageSize);
  const client = new EC2Client({});

  // The paginate function is a wrapper around the underlying command.
  const paginator = paginateDescribeInstanceTypes(
    // Without limiting the page size, this call can take a long time. pageSize is
    just sugar for
    // the MaxResults property in the underlying command.
    { client, pageSize },
    {
      Filters: [
        {
          Name: "processor-info.supported-architecture",
          Values: supportedArch,
        },
        { Name: "free-tier-eligible", Values: [freeTier ? "true" : "false"] },
      ],
    },
  );
};
```

```
    ],
  },
);

try {
  /**
   * @type {import('@aws-sdk/client-ec2').InstanceTypeInfo[]}
   */
  const instanceTypes = [];

  for await (const page of paginator) {
    if (page.InstanceTypes.length) {
      instanceTypes.push(...page.InstanceTypes);

      // When we have at least 1 result, we can stop.
      if (instanceTypes.length >= 1) {
        break;
      }
    }
  }
  console.log(
    `Memory size in MiB for matching instance types:\n\n${instanceTypes.map((it)
=> `${it.InstanceType}: ${it.MemoryInfo.SizeInMiB} MiB`).join("\n")}`,
  );
} catch (caught) {
  if (caught instanceof Error && caught.name === "InvalidParameterValue") {
    console.warn(`${caught.message}`);
    return [];
  }
  throw caught;
}
};
```

- For API details, see [DescribeInstanceTypes](#) in *AWS SDK for JavaScript API Reference*.

DescribeInstances

The following code example shows how to use `DescribeInstances`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { EC2Client, paginateDescribeInstances } from "@aws-sdk/client-ec2";

/**
 * List all of your EC2 instances running with the provided architecture that
 * were launched in the past month.
 * @param {{ pageSize: string, architectures: string[] }} options
 */
export const main = async ({ pageSize, architectures }) => {
  pageSize = parseInt(pageSize);
  const client = new EC2Client({});
  const d = new Date();
  const year = d.getFullYear();
  const month = `0${d.getMonth() + 1}`.slice(-2);
  const launchTimePattern = `${year}-${month}-*`;

  const paginator = paginateDescribeInstances(
    {
      client,
      pageSize,
    },
    {
      Filters: [
        { Name: "architecture", Values: architectures },
        { Name: "instance-state-name", Values: ["running"] },
        {
          Name: "launch-time",
          Values: [launchTimePattern],
        },
      ],
    },
  );

  try {
    /**
```

```

    * @type {import('@aws-sdk/client-ec2').Instance[]}
    */
    const instanceList = [];
    for await (const page of paginator) {
      const { Reservations } = page;
      Reservations.forEach((r) => instanceList.push(...r.Instances));
    }
    console.log(
      `Running instances launched this month:\n\n${instanceList.map((instance) =>
instance.InstanceId).join("\n")}`,
    );
  } catch (caught) {
    if (caught instanceof Error && caught.name === "InvalidParameterValue") {
      console.warn(`${caught.message}.`);
    } else {
      throw caught;
    }
  }
};

```

- For API details, see [DescribeInstances](#) in *AWS SDK for JavaScript API Reference*.

DescribeKeyPairs

The following code example shows how to use `DescribeKeyPairs`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```

import { DescribeKeyPairsCommand, EC2Client } from "@aws-sdk/client-ec2";

/**
 * List all key pairs in the current AWS account.
 * @param {{ dryRun: boolean }}
 */

```

```
export const main = async ({ dryRun }) => {
  const client = new EC2Client({});
  const command = new DescribeKeyPairsCommand({ DryRun: dryRun });

  try {
    const { KeyPairs } = await client.send(command);
    const keyPairList = KeyPairs.map(
      (kp) => ` • ${kp.KeyPairId}: ${kp.KeyName}`,
    ).join("\n");
    console.log("The following key pairs were found in your account:");
    console.log(keyPairList);
  } catch (caught) {
    if (caught instanceof Error && caught.name === "DryRunOperation") {
      console.log(`${caught.message}`);
    } else {
      throw caught;
    }
  }
};
```

- For API details, see [DescribeKeyPairs](#) in *AWS SDK for JavaScript API Reference*.

DescribeRegions

The following code example shows how to use DescribeRegions.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { DescribeRegionsCommand, EC2Client } from "@aws-sdk/client-ec2";

/**
 * List all available AWS regions.
 * @param {{ regionNames: string[], includeOptInRegions: boolean }} options
 */
```



```
export const main = async ({ regionNames, includeOptInRegions }) => {
  const client = new EC2Client({});
  const command = new DescribeRegionsCommand({
    // By default this command will not show regions that require you to opt-in.
    // When AllRegions is true, even the regions that require opt-in will be
    returned.
    AllRegions: includeOptInRegions,
    // You can omit the Filters property if you want to get all regions.
    Filters: regionNames?.length
      ? [
        {
          Name: "region-name",
          // You can specify multiple values for a filter.
          // You can also use '*' as a wildcard. This will return all
          // of the regions that start with `us-east-`.
          Values: regionNames,
        },
      ]
      : undefined,
  });

  try {
    const { Regions } = await client.send(command);
    const regionsList = Regions.map((reg) => ` • ${reg.RegionName}`);
    console.log("Found regions:");
    console.log(regionsList.join("\n"));
  } catch (caught) {
    if (caught instanceof Error && caught.name === "DryRunOperation") {
      console.log(`${caught.message}`);
    } else {
      throw caught;
    }
  }
};
```

- For API details, see [DescribeRegions](#) in *AWS SDK for JavaScript API Reference*.

DescribeSecurityGroups

The following code example shows how to use DescribeSecurityGroups.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { DescribeSecurityGroupsCommand, EC2Client } from "@aws-sdk/client-ec2";

/**
 * Describes the specified security groups or all of your security groups.
 * @param {{ groupIds: string[] }} options
 */
export const main = async ({ groupIds = [] }) => {
  const client = new EC2Client({});
  const command = new DescribeSecurityGroupsCommand({
    GroupIds: groupIds,
  });

  try {
    const { SecurityGroups } = await client.send(command);
    const sgList = SecurityGroups.map(
      (sg) => `• ${sg.GroupName} (${sg.GroupId}): ${sg.Description}`,
    ).join("\n");
    if (sgList.length) {
      console.log(`Security groups:\n${sgList}`);
    } else {
      console.log("No security groups found.");
    }
  } catch (caught) {
    if (caught instanceof Error && caught.name === "InvalidGroupId.Malformed") {
      console.warn(`${caught.message}. Please provide a valid GroupId.`);
    } else if (
      caught instanceof Error &&
      caught.name === "InvalidGroup.NotFound"
    ) {
      console.warn(caught.message);
    } else {
      throw caught;
    }
  }
}
```

```
};
```

- For API details, see [DescribeSecurityGroups](#) in *AWS SDK for JavaScript API Reference*.

DescribeSubnets

The following code example shows how to use `DescribeSubnets`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const client = new EC2Client({});
const { Subnets } = await client.send(
  new DescribeSubnetsCommand({
    Filters: [
      { Name: "vpc-id", Values: [state.defaultVpc] },
      { Name: "availability-zone", Values: state.availabilityZoneNames },
      { Name: "default-for-az", Values: ["true"] },
    ],
  }),
);
```

- For API details, see [DescribeSubnets](#) in *AWS SDK for JavaScript API Reference*.

DescribeVpcs

The following code example shows how to use `DescribeVpcs`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const client = new EC2Client({});
const { Vpcs } = await client.send(
  new DescribeVpcsCommand({
    Filters: [{ Name: "is-default", Values: ["true"] }],
  }),
);
```

- For API details, see [DescribeVpcs](#) in *AWS SDK for JavaScript API Reference*.

DisassociateAddress

The following code example shows how to use `DisassociateAddress`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { DisassociateAddressCommand, EC2Client } from "@aws-sdk/client-ec2";

/**
 * Disassociate an Elastic IP address from an instance.
 * @param {{ associationId: string }} options
 */
export const main = async ({ associationId }) => {
  const client = new EC2Client({});
  const command = new DisassociateAddressCommand({
```

```
// You can also use PublicIp, but that is for EC2 classic which is being
retired.
  AssociationId: associationId,
});

try {
  await client.send(command);
  console.log("Successfully disassociated address");
} catch (caught) {
  if (
    caught instanceof Error &&
    caught.name === "InvalidAssociationID.NotFound"
  ) {
    console.warn(`${caught.message}.`);
  } else {
    throw caught;
  }
}
};
```

- For API details, see [DisassociateAddress](#) in *AWS SDK for JavaScript API Reference*.

MonitorInstances

The following code example shows how to use MonitorInstances.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { EC2Client, MonitorInstancesCommand } from "@aws-sdk/client-ec2";

/**
 * Turn on detailed monitoring for the selected instance.
 * By default, metrics are sent to Amazon CloudWatch every 5 minutes.
 * For a cost you can enable detailed monitoring which sends metrics every minute.
 * @param {{ instanceIds: string[] }} options
```

```
*/
export const main = async ({ instanceIds }) => {
  const client = new EC2Client({});
  const command = new MonitorInstancesCommand({
    InstanceIds: instanceIds,
  });

  try {
    const { InstanceMonitorings } = await client.send(command);
    const instancesBeingMonitored = InstanceMonitorings.map(
      (im) =>
        ` • Detailed monitoring state for ${im.InstanceId} is
${im.Monitoring.State}.`,
    );
    console.log("Monitoring status:");
    console.log(instancesBeingMonitored.join("\n"));
  } catch (caught) {
    if (caught instanceof Error && caught.name === "InvalidParameterValue") {
      console.warn(` ${caught.message} `);
    } else {
      throw caught;
    }
  }
};
```

- For API details, see [MonitorInstances](#) in *AWS SDK for JavaScript API Reference*.

RebootInstances

The following code example shows how to use `RebootInstances`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { EC2Client, RebootInstancesCommand } from "@aws-sdk/client-ec2";
```

```
/**
 * Requests a reboot of the specified instances. This operation is asynchronous;
 * it only queues a request to reboot the specified instances.
 * @param {{ instanceIds: string[] }} options
 */
export const main = async ({ instanceIds }) => {
  const client = new EC2Client({});
  const command = new RebootInstancesCommand({
    InstanceIds: instanceIds,
  });

  try {
    await client.send(command);
    console.log("Instance rebooted successfully.");
  } catch (caught) {
    if (
      caught instanceof Error &&
      caught.name === "InvalidInstanceID.NotFound"
    ) {
      console.warn(
        `${caught.message}. Please provide the InstanceId of a valid instance to
reboot.`
      );
    } else {
      throw caught;
    }
  }
};
```

- For API details, see [RebootInstances](#) in *AWS SDK for JavaScript API Reference*.

ReleaseAddress

The following code example shows how to use ReleaseAddress.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { ReleaseAddressCommand, EC2Client } from "@aws-sdk/client-ec2";

/**
 * Release an Elastic IP address.
 * @param {{ allocationId: string }} options
 */
export const main = async ({ allocationId }) => {
  const client = new EC2Client({});
  const command = new ReleaseAddressCommand({
    // You can also use PublicIp, but that is for EC2 classic which is being
    retired.
    AllocationId: allocationId,
  });

  try {
    await client.send(command);
    console.log("Successfully released address.");
  } catch (caught) {
    if (
      caught instanceof Error &&
      caught.name === "InvalidAllocationID.NotFound"
    ) {
      console.warn(`${caught.message}. Please provide a valid AllocationID.`);
    } else {
      throw caught;
    }
  }
};
```

- For API details, see [ReleaseAddress](#) in *AWS SDK for JavaScript API Reference*.

ReplaceIamInstanceProfileAssociation

The following code example shows how to use `ReplaceIamInstanceProfileAssociation`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
await retry({ intervalInMs: 1000, maxRetries: 30 }, () =>
  ec2Client.send(
    new ReplaceIamInstanceProfileAssociationCommand({
      AssociationId: state.instanceProfileAssociationId,
      IamInstanceProfile: { Name: NAMES.ssmOnlyInstanceProfileName },
    }),
  ),
);
```

- For API details, see [ReplacelamInstanceProfileAssociation](#) in *AWS SDK for JavaScript API Reference*.

RunInstances

The following code example shows how to use RunInstances.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { EC2Client, RunInstancesCommand } from "@aws-sdk/client-ec2";

/**
 * Create new EC2 instances.
 * @param {{
 *   keyName: string,
 *   securityGroupIds: string[],
```

```
*   imageId: string,
*   instanceType: import('@aws-sdk/client-ec2')._InstanceType,
*   minCount?: number,
*   maxCount?: number }} options
*/
export const main = async ({
  keyName,
  securityGroupIds,
  imageId,
  instanceType,
  minCount = "1",
  maxCount = "1",
}) => {
  const client = new EC2Client({});
  minCount = parseInt(minCount);
  maxCount = parseInt(maxCount);
  const command = new RunInstancesCommand({
    // Your key pair name.
    KeyName: keyName,
    // Your security group.
    SecurityGroupIds: securityGroupIds,
    // An Amazon Machine Image (AMI). There are multiple ways to search for AMIs.
    // For more information, see:
    // https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/finding-an-ami.html
    ImageId: imageId,
    // An instance type describing the resources provided to your instance. There
    // are multiple
    // ways to search for instance types. For more information see:
    // https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-discovery.html
    InstanceType: instanceType,
    // Availability Zones have capacity limitations that may impact your ability to
    // launch instances.
    // The `RunInstances` operation will only succeed if it can allocate at least
    // the `MinCount` of instances.
    // However, EC2 will attempt to launch up to the `MaxCount` of instances, even
    // if the full request cannot be satisfied.
    // If you need a specific number of instances, use `MinCount` and `MaxCount` set
    // to the same value.
    // If you want to launch up to a certain number of instances, use `MaxCount` and
    // let EC2 provision as many as possible.
    // If you require a minimum number of instances, but do not want to exceed a
    // maximum, use both `MinCount` and `MaxCount`.
    MinCount: minCount,
    MaxCount: maxCount,
```

```
});

try {
  const { Instances } = await client.send(command);
  const instanceList = Instances.map(
    (instance) => `• ${instance.InstanceId}`,
  ).join("\n");
  console.log(`Launched instances:\n${instanceList}`);
} catch (caught) {
  if (caught instanceof Error && caught.name === "ResourceCountExceeded") {
    console.warn(`${caught.message}`);
  } else {
    throw caught;
  }
}
};
```

- For API details, see [RunInstances](#) in *AWS SDK for JavaScript API Reference*.

StartInstances

The following code example shows how to use StartInstances.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { EC2Client, StartInstancesCommand } from "@aws-sdk/client-ec2";
import { fileURLToPath } from "url";
import { parseArgs } from "util";

/**
 * Starts an Amazon EBS-backed instance that you've previously stopped.
 * @param {{ instanceIds }} options
 */
export const main = async ({ instanceIds }) => {
  const client = new EC2Client({});
```

```
const command = new StartInstancesCommand({
  InstanceIds: instanceIds,
});

try {
  const { StartingInstances } = await client.send(command);
  const instanceIdList = StartingInstances.map(
    (instance) => ` • ${instance.InstanceId}`,
  );
  console.log("Starting instances:");
  console.log(instanceIdList.join("\n"));
} catch (caught) {
  if (
    caught instanceof Error &&
    caught.name === "InvalidInstanceID.NotFound"
  ) {
    console.warn(`${caught.message}`);
  } else {
    throw caught;
  }
}
};
```

- For API details, see [StartInstances](#) in *AWS SDK for JavaScript API Reference*.

StopInstances

The following code example shows how to use StopInstances.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { EC2Client, StopInstancesCommand } from "@aws-sdk/client-ec2";
import { fileURLToPath } from "url";
import { parseArgs } from "util";
```

```
/**
 * Stop one or more EC2 instances.
 * @param {{ instanceIds: string[] }} options
 */
export const main = async ({ instanceIds }) => {
  const client = new EC2Client({});
  const command = new StopInstancesCommand({
    InstanceIds: instanceIds,
  });

  try {
    const { StoppingInstances } = await client.send(command);
    const instanceIdList = StoppingInstances.map(
      (instance) => ` • ${instance.InstanceId}`,
    );
    console.log("Stopping instances:");
    console.log(instanceIdList.join("\n"));
  } catch (caught) {
    if (
      caught instanceof Error &&
      caught.name === "InvalidInstanceID.NotFound"
    ) {
      console.warn(`${caught.message}`);
    } else {
      throw caught;
    }
  }
};
```

- For API details, see [StopInstances](#) in *AWS SDK for JavaScript API Reference*.

TerminateInstances

The following code example shows how to use `TerminateInstances`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { EC2Client, TerminateInstancesCommand } from "@aws-sdk/client-ec2";
import { fileURLToPath } from "url";
import { parseArgs } from "util";

/**
 * Terminate one or more EC2 instances.
 * @param {{ instanceIds: string[] }} options
 */
export const main = async ({ instanceIds }) => {
  const client = new EC2Client({});
  const command = new TerminateInstancesCommand({
    InstanceIds: instanceIds,
  });

  try {
    const { TerminatingInstances } = await client.send(command);
    const instanceList = TerminatingInstances.map(
      (instance) => ` • ${instance.InstanceId}`,
    );
    console.log("Terminating instances:");
    console.log(instanceList.join("\n"));
  } catch (caught) {
    if (
      caught instanceof Error &&
      caught.name === "InvalidInstanceID.NotFound"
    ) {
      console.warn(` ${caught.message} `);
    } else {
      throw caught;
    }
  }
};
```

- For API details, see [TerminateInstances](#) in *AWS SDK for JavaScript API Reference*.

UnmonitorInstances

The following code example shows how to use `UnmonitorInstances`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { EC2Client, UnmonitorInstancesCommand } from "@aws-sdk/client-ec2";
import { fileURLToPath } from "url";
import { parseArgs } from "util";

/**
 * Turn off detailed monitoring for the selected instance.
 * @param {{ instanceIds: string[] }} options
 */
export const main = async ({ instanceIds }) => {
  const client = new EC2Client({});
  const command = new UnmonitorInstancesCommand({
    InstanceIds: instanceIds,
  });

  try {
    const { InstanceMonitorings } = await client.send(command);
    const instanceMonitoringsList = InstanceMonitorings.map(
      (im) =>
        ` • Detailed monitoring state for ${im.InstanceId} is
        ${im.Monitoring.State}.`,
    );
    console.log("Monitoring status:");
    console.log(instanceMonitoringsList.join("\n"));
  } catch (caught) {
    if (
      caught instanceof Error &&
      caught.name === "InvalidInstanceID.NotFound"
    ) {
      console.warn(` ${caught.message} `);
    } else {
      throw caught;
    }
  }
};
```

- For API details, see [UnmonitorInstances](#) in *AWS SDK for JavaScript API Reference*.

Scenarios

Build and manage a resilient service

The following code example shows how to create a load-balanced web service that returns book, movie, and song recommendations. The example shows how the service responds to failures, and how to restructure the service for more resilience when failures occur.

- Use an Amazon EC2 Auto Scaling group to create Amazon Elastic Compute Cloud (Amazon EC2) instances based on a launch template and to keep the number of instances in a specified range.
- Handle and distribute HTTP requests with Elastic Load Balancing.
- Monitor the health of instances in an Auto Scaling group and forward requests only to healthy instances.
- Run a Python web server on each EC2 instance to handle HTTP requests. The web server responds with recommendations and health checks.
- Simulate a recommendation service with an Amazon DynamoDB table.
- Control web server response to requests and health checks by updating AWS Systems Manager parameters.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Run the interactive scenario at a command prompt.

```
#!/usr/bin/env node
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
```



```
import {
  Scenario,
  parseScenarioArgs,
} from "@aws-doc-sdk-examples/lib/scenario/index.js";

/**
 * The workflow steps are split into three stages:
 * - deploy
 * - demo
 * - destroy
 *
 * Each of these stages has a corresponding file prefixed with steps-*.
 */
import { deploySteps } from "./steps-deploy.js";
import { demoSteps } from "./steps-demo.js";
import { destroySteps } from "./steps-destroy.js";

/**
 * The context is passed to every scenario. Scenario steps
 * will modify the context.
 */
const context = {};

/**
 * Three Scenarios are created for the workflow. A Scenario is an orchestration
 class
 * that simplifies running a series of steps.
 */
export const scenarios = {
  // Deploys all resources necessary for the workflow.
  deploy: new Scenario("Resilient Workflow - Deploy", deploySteps, context),
  // Demonstrates how a fragile web service can be made more resilient.
  demo: new Scenario("Resilient Workflow - Demo", demoSteps, context),
  // Destroys the resources created for the workflow.
  destroy: new Scenario("Resilient Workflow - Destroy", destroySteps, context),
};

// Call function if run directly
import { fileURLToPath } from "url";

if (process.argv[1] === fileURLToPath(import.meta.url)) {
  parseScenarioArgs(scenarios);
}
```

Create steps to deploy all of the resources.

```
import { join } from "node:path";
import { readFileSync, writeFileSync } from "node:fs";
import axios from "axios";

import {
  BatchWriteItemCommand,
  CreateTableCommand,
  DynamoDBClient,
  waitUntilTableExists,
} from "@aws-sdk/client-dynamodb";
import {
  EC2Client,
  CreateKeyPairCommand,
  CreateLaunchTemplateCommand,
  DescribeAvailabilityZonesCommand,
  DescribeVpcsCommand,
  DescribeSubnetsCommand,
  DescribeSecurityGroupsCommand,
  AuthorizeSecurityGroupIngressCommand,
} from "@aws-sdk/client-ec2";
import {
  IAMClient,
  CreatePolicyCommand,
  CreateRoleCommand,
  CreateInstanceProfileCommand,
  AddRoleToInstanceProfileCommand,
  AttachRolePolicyCommand,
  waitUntilInstanceProfileExists,
} from "@aws-sdk/client-iam";
import { SSMClient, GetParameterCommand } from "@aws-sdk/client-ssm";
import {
  CreateAutoScalingGroupCommand,
  AutoScalingClient,
  AttachLoadBalancerTargetGroupsCommand,
} from "@aws-sdk/client-auto-scaling";
import {
  CreateListenerCommand,
  CreateLoadBalancerCommand,
  CreateTargetGroupCommand,
```

```
ElasticLoadBalancingV2Client,
waitUntilLoadBalancerAvailable,
} from "@aws-sdk/client-elastic-load-balancing-v2";

import {
  ScenarioOutput,
  ScenarioInput,
  ScenarioAction,
} from "@aws-doc-sdk-examples/lib/scenario/index.js";
import { saveState } from "@aws-doc-sdk-examples/lib/scenario/steps-common.js";
import { retry } from "@aws-doc-sdk-examples/lib/utils/util-timers.js";

import { MESSAGES, NAMES, RESOURCES_PATH, ROOT } from "./constants.js";
import { initParamsSteps } from "./steps-reset-params.js";

/**
 * @type {import('@aws-doc-sdk-examples/lib/scenario.js').Step[]}
 */
export const deploySteps = [
  new ScenarioOutput("introduction", MESSAGES.introduction, { header: true }),
  new ScenarioInput("confirmDeployment", MESSAGES.confirmDeployment, {
    type: "confirm",
  }),
  new ScenarioAction(
    "handleConfirmDeployment",
    (c) => c.confirmDeployment === false && process.exit(),
  ),
  new ScenarioOutput(
    "creatingTable",
    MESSAGES.creatingTable.replace("${TABLE_NAME}", NAMES.tableName),
  ),
  new ScenarioAction("createTable", async () => {
    const client = new DynamoDBClient({});
    await client.send(
      new CreateTableCommand({
        TableName: NAMES.tableName,
        ProvisionedThroughput: {
          ReadCapacityUnits: 5,
          WriteCapacityUnits: 5,
        },
        AttributeDefinitions: [
          {
            AttributeName: "MediaType",
            AttributeType: "S",
          }
        ]
      })
    );
  })
];
```

```

    },
    {
      AttributeName: "ItemId",
      AttributeType: "N",
    },
  ],
  KeySchema: [
    {
      AttributeName: "MediaType",
      KeyType: "HASH",
    },
    {
      AttributeName: "ItemId",
      KeyType: "RANGE",
    },
  ],
}),
);
await waitUntilTableExists({ client }, { TableName: NAMES.tableName });
}),
new ScenarioOutput(
  "createdTable",
  MESSAGES.createdTable.replace("${TABLE_NAME}", NAMES.tableName),
),
new ScenarioOutput(
  "populatingTable",
  MESSAGES.populatingTable.replace("${TABLE_NAME}", NAMES.tableName),
),
new ScenarioAction("populateTable", () => {
  const client = new DynamoDBClient({});
  /**
   * @type {{ default: import("@aws-sdk/client-dynamodb").PutRequest['Item'][] }}
   */
  const recommendations = JSON.parse(
    readFileSync(join(RESOURCES_PATH, "recommendations.json")),
  );

  return client.send(
    new BatchWriteItemCommand({
      RequestItems: {
        [NAMES.tableName]: recommendations.map((item) => ({
          PutRequest: { Item: item },
        })),
      },
    }),
  ),
},

```

```
    }),
  );
}),
new ScenarioOutput(
  "populatedTable",
  MESSAGES.populatedTable.replace("${TABLE_NAME}", NAMES.tableName),
),
new ScenarioOutput(
  "creatingKeyPair",
  MESSAGES.creatingKeyPair.replace("${KEY_PAIR_NAME}", NAMES.keyPairName),
),
new ScenarioAction("createKeyPair", async () => {
  const client = new EC2Client({});
  const { KeyMaterial } = await client.send(
    new CreateKeyPairCommand({
      KeyName: NAMES.keyPairName,
    }),
  );
  writeFileSync(`${NAMES.keyPairName}.pem`, KeyMaterial, { mode: 0o600 });
}),
new ScenarioOutput(
  "createdKeyPair",
  MESSAGES.createdKeyPair.replace("${KEY_PAIR_NAME}", NAMES.keyPairName),
),
new ScenarioOutput(
  "creatingInstancePolicy",
  MESSAGES.creatingInstancePolicy.replace(
    "${INSTANCE_POLICY_NAME}",
    NAMES.instancePolicyName,
  ),
),
new ScenarioAction("createInstancePolicy", async (state) => {
  const client = new IAMClient({});
  const {
    Policy: { Arn },
  } = await client.send(
    new CreatePolicyCommand({
      PolicyName: NAMES.instancePolicyName,
      PolicyDocument: readFileSync(
        join(RESOURCES_PATH, "instance_policy.json"),
      ),
    }),
  );
});
```

```
    state.instancePolicyArn = Arn;
  }),
  new ScenarioOutput("createdInstancePolicy", (state) =>
    MESSAGES.createdInstancePolicy
      .replace("${INSTANCE_POLICY_NAME}", NAMES.instancePolicyName)
      .replace("${INSTANCE_POLICY_ARN}", state.instancePolicyArn),
  ),
  new ScenarioOutput(
    "creatingInstanceRole",
    MESSAGES.creatingInstanceRole.replace(
      "${INSTANCE_ROLE_NAME}",
      NAMES.instanceRoleName,
    ),
  ),
  new ScenarioAction("createInstanceRole", () => {
    const client = new IAMClient({});
    return client.send(
      new CreateRoleCommand({
        RoleName: NAMES.instanceRoleName,
        AssumeRolePolicyDocument: readFileSync(
          join(ROOT, "assume-role-policy.json"),
        ),
      }),
    ),
  }),
  new ScenarioOutput(
    "createdInstanceRole",
    MESSAGES.createdInstanceRole.replace(
      "${INSTANCE_ROLE_NAME}",
      NAMES.instanceRoleName,
    ),
  ),
  new ScenarioOutput(
    "attachingPolicyToRole",
    MESSAGES.attachingPolicyToRole
      .replace("${INSTANCE_ROLE_NAME}", NAMES.instanceRoleName)
      .replace("${INSTANCE_POLICY_NAME}", NAMES.instancePolicyName),
  ),
  new ScenarioAction("attachPolicyToRole", async (state) => {
    const client = new IAMClient({});
    await client.send(
      new AttachRolePolicyCommand({
        RoleName: NAMES.instanceRoleName,
        PolicyArn: state.instancePolicyArn,
```

```
    }),
  );
}),
new ScenarioOutput(
  "attachedPolicyToRole",
  MESSAGES.attachedPolicyToRole
    .replace("${INSTANCE_POLICY_NAME}", NAMES.instancePolicyName)
    .replace("${INSTANCE_ROLE_NAME}", NAMES.instanceRoleName),
),
new ScenarioOutput(
  "creatingInstanceProfile",
  MESSAGES.creatingInstanceProfile.replace(
    "${INSTANCE_PROFILE_NAME}",
    NAMES.instanceProfileName,
  ),
),
new ScenarioAction("createInstanceProfile", async (state) => {
  const client = new IAMClient({});
  const {
    InstanceProfile: { Arn },
  } = await client.send(
    new CreateInstanceProfileCommand({
      InstanceProfileName: NAMES.instanceProfileName,
    }),
  );
  state.instanceProfileArn = Arn;

  await waitUntilInstanceProfileExists(
    { client },
    { InstanceProfileName: NAMES.instanceProfileName },
  );
}),
new ScenarioOutput("createdInstanceProfile", (state) =>
  MESSAGES.createdInstanceProfile
    .replace("${INSTANCE_PROFILE_NAME}", NAMES.instanceProfileName)
    .replace("${INSTANCE_PROFILE_ARN}", state.instanceProfileArn),
),
new ScenarioOutput(
  "addingRoleToInstanceProfile",
  MESSAGES.addingRoleToInstanceProfile
    .replace("${INSTANCE_PROFILE_NAME}", NAMES.instanceProfileName)
    .replace("${INSTANCE_ROLE_NAME}", NAMES.instanceRoleName),
),
new ScenarioAction("addRoleToInstanceProfile", () => {
```

```
const client = new IAMClient({});
return client.send(
  new AddRoleToInstanceProfileCommand({
    RoleName: NAMES.instanceRoleName,
    InstanceProfileName: NAMES.instanceProfileName,
  }),
);
}),
new ScenarioOutput(
  "addedRoleToInstanceProfile",
  MESSAGES.addedRoleToInstanceProfile
    .replace("${INSTANCE_PROFILE_NAME}", NAMES.instanceProfileName)
    .replace("${INSTANCE_ROLE_NAME}", NAMES.instanceRoleName),
),
...initParamsSteps,
new ScenarioOutput("creatingLaunchTemplate", MESSAGES.creatingLaunchTemplate),
new ScenarioAction("createLaunchTemplate", async () => {
  const ssmClient = new SSMClient({});
  const { Parameter } = await ssmClient.send(
    new GetParameterCommand({
      Name: "/aws/service/ami-amazon-linux-latest/amzn2-ami-hvm-x86_64-gp2",
    }),
  );
  const ec2Client = new EC2Client({});
  await ec2Client.send(
    new CreateLaunchTemplateCommand({
      LaunchTemplateName: NAMES.launchTemplateName,
      LaunchTemplateData: {
        InstanceType: "t3.micro",
        ImageId: Parameter.Value,
        IamInstanceProfile: { Name: NAMES.instanceProfileName },
        UserData: readFileSync(
          join(RESOURCES_PATH, "server_startup_script.sh"),
        ).toString("base64"),
        KeyName: NAMES.keyPairName,
      },
    }),
  );
}),
new ScenarioOutput(
  "createdLaunchTemplate",
  MESSAGES.createdLaunchTemplate.replace(
    "${LAUNCH_TEMPLATE_NAME}",
    NAMES.launchTemplateName,
  ),

```



```

    ),
  ),
  new ScenarioOutput(
    "creatingAutoScalingGroup",
    MESSAGES.creatingAutoScalingGroup.replace(
      "${AUTO_SCALING_GROUP_NAME}",
      NAMES.autoScalingGroupName,
    ),
  ),
  new ScenarioAction("createAutoScalingGroup", async (state) => {
    const ec2Client = new EC2Client({});
    const { AvailabilityZones } = await ec2Client.send(
      new DescribeAvailabilityZonesCommand({}),
    );
    state.availabilityZoneNames = AvailabilityZones.map((az) => az.ZoneName);
    const autoScalingClient = new AutoScalingClient({});
    await retry({ intervalInMs: 1000, maxRetries: 30 }, () =>
      autoScalingClient.send(
        new CreateAutoScalingGroupCommand({
          AvailabilityZones: state.availabilityZoneNames,
          AutoScalingGroupName: NAMES.autoScalingGroupName,
          LaunchTemplate: {
            LaunchTemplateName: NAMES.launchTemplateName,
            Version: "$Default",
          },
          MinSize: 3,
          MaxSize: 3,
        }),
      ),
    );
  }),
  new ScenarioOutput(
    "createdAutoScalingGroup",
    /**
     * @param {{ availabilityZoneNames: string[] }} state
     */
    (state) =>
      MESSAGES.createdAutoScalingGroup
        .replace("${AUTO_SCALING_GROUP_NAME}", NAMES.autoScalingGroupName)
        .replace(
          "${AVAILABILITY_ZONE_NAMES}",
          state.availabilityZoneNames.join(", "),
        ),
  ),
),

```

```

new ScenarioInput("confirmContinue", MESSAGES.confirmContinue, {
  type: "confirm",
}),
new ScenarioOutput("loadBalancer", MESSAGES.loadBalancer),
new ScenarioOutput("gettingVpc", MESSAGES.gettingVpc),
new ScenarioAction("getVpc", async (state) => {
  const client = new EC2Client({});
  const { Vpcs } = await client.send(
    new DescribeVpcsCommand({
      Filters: [{ Name: "is-default", Values: ["true"] }],
    }),
  );
  state.defaultVpc = Vpcs[0].VpcId;
}),
new ScenarioOutput("gotVpc", (state) =>
  MESSAGES.gotVpc.replace("${VPC_ID}", state.defaultVpc),
),
new ScenarioOutput("gettingSubnets", MESSAGES.gettingSubnets),
new ScenarioAction("getSubnets", async (state) => {
  const client = new EC2Client({});
  const { Subnets } = await client.send(
    new DescribeSubnetsCommand({
      Filters: [
        { Name: "vpc-id", Values: [state.defaultVpc] },
        { Name: "availability-zone", Values: state.availabilityZoneNames },
        { Name: "default-for-az", Values: ["true"] },
      ],
    }),
  );
  state.subnets = Subnets.map((subnet) => subnet.SubnetId);
}),
new ScenarioOutput(
  "gotSubnets",
  /**
   * @param {{ subnets: string[] }} state
   */
  (state) =>
    MESSAGES.gotSubnets.replace("${SUBNETS}", state.subnets.join(", ")),
),
new ScenarioOutput(
  "creatingLoadBalancerTargetGroup",
  MESSAGES.creatingLoadBalancerTargetGroup.replace(
    "${TARGET_GROUP_NAME}",
    NAMES.loadBalancerTargetGroupName,

```

```

    ),
  ),
  new ScenarioAction("createLoadBalancerTargetGroup", async (state) => {
    const client = new ElasticLoadBalancingV2Client({});
    const { TargetGroups } = await client.send(
      new CreateTargetGroupCommand({
        Name: NAMES.loadBalancerTargetGroupName,
        Protocol: "HTTP",
        Port: 80,
        HealthCheckPath: "/healthcheck",
        HealthCheckIntervalSeconds: 10,
        HealthCheckTimeoutSeconds: 5,
        HealthyThresholdCount: 2,
        UnhealthyThresholdCount: 2,
        VpcId: state.defaultVpc,
      })
    ),
  );
  const targetGroup = TargetGroups[0];
  state.targetGroupArn = targetGroup.TargetGroupArn;
  state.targetGroupProtocol = targetGroup.Protocol;
  state.targetGroupPort = targetGroup.Port;
}),
  new ScenarioOutput(
    "createdLoadBalancerTargetGroup",
    MESSAGES.createdLoadBalancerTargetGroup.replace(
      "${TARGET_GROUP_NAME}",
      NAMES.loadBalancerTargetGroupName,
    ),
  ),
  new ScenarioOutput(
    "creatingLoadBalancer",
    MESSAGES.creatingLoadBalancer.replace("${LB_NAME}", NAMES.loadBalancerName),
  ),
  new ScenarioAction("createLoadBalancer", async (state) => {
    const client = new ElasticLoadBalancingV2Client({});
    const { LoadBalancers } = await client.send(
      new CreateLoadBalancerCommand({
        Name: NAMES.loadBalancerName,
        Subnets: state.subnets,
      })
    ),
  );
  state.loadBalancerDns = LoadBalancers[0].DNSName;
  state.loadBalancerArn = LoadBalancers[0].LoadBalancerArn;
  await waitUntilLoadBalancerAvailable(

```

```
    { client },
    { Names: [NAMES.loadBalancerName] },
  );
}),
new ScenarioOutput("createdLoadBalancer", (state) =>
  MESSAGES.createdLoadBalancer
    .replace("${LB_NAME}", NAMES.loadBalancerName)
    .replace("${DNS_NAME}", state.loadBalancerDns),
),
new ScenarioOutput(
  "creatingListener",
  MESSAGES.creatingLoadBalancerListener
    .replace("${LB_NAME}", NAMES.loadBalancerName)
    .replace("${TARGET_GROUP_NAME}", NAMES.loadBalancerTargetGroupName),
),
new ScenarioAction("createListener", async (state) => {
  const client = new ElasticLoadBalancingV2Client({});
  const { Listeners } = await client.send(
    new CreateListenerCommand({
      LoadBalancerArn: state.loadBalancerArn,
      Protocol: state.targetGroupProtocol,
      Port: state.targetGroupPort,
      DefaultActions: [
        { Type: "forward", TargetGroupArn: state.targetGroupArn },
      ],
    })
  );
  const listener = Listeners[0];
  state.loadBalancerListenerArn = listener.ListenerArn;
}),
new ScenarioOutput("createdListener", (state) =>
  MESSAGES.createdLoadBalancerListener.replace(
    "${LB_LISTENER_ARN}",
    state.loadBalancerListenerArn,
  ),
),
new ScenarioOutput(
  "attachingLoadBalancerTargetGroup",
  MESSAGES.attachingLoadBalancerTargetGroup
    .replace("${TARGET_GROUP_NAME}", NAMES.loadBalancerTargetGroupName)
    .replace("${AUTO_SCALING_GROUP_NAME}", NAMES.autoScalingGroupName),
),
new ScenarioAction("attachLoadBalancerTargetGroup", async (state) => {
  const client = new AutoScalingClient({});
```

```

    await client.send(
      new AttachLoadBalancerTargetGroupsCommand({
        AutoScalingGroupName: NAMES.autoScalingGroupName,
        TargetGroupARNs: [state.targetGroupArn],
      }),
    );
  }},
  new ScenarioOutput(
    "attachedLoadBalancerTargetGroup",
    MESSAGES.attachedLoadBalancerTargetGroup,
  ),
  new ScenarioOutput("verifyingInboundPort", MESSAGES.verifyingInboundPort),
  new ScenarioAction(
    "verifyInboundPort",
    /**
     *
     * @param {{ defaultSecurityGroup: import('@aws-sdk/client-ec2').SecurityGroup}}
state
     */
    async (state) => {
      const client = new EC2Client({});
      const { SecurityGroups } = await client.send(
        new DescribeSecurityGroupsCommand({
          Filters: [{ Name: "group-name", Values: ["default"] }]},
        ),
      );
      if (!SecurityGroups) {
        state.verifyInboundPortError = new Error(MESSAGES.noSecurityGroups);
      }
      state.defaultSecurityGroup = SecurityGroups[0];

      /**
       * @type {string}
       */
      const ipResponse = (await axios.get("http://checkip.amazonaws.com")).data;
      state.myIp = ipResponse.trim();
      const myIpRules = state.defaultSecurityGroup.IpPermissions.filter(
        ({ IpRanges }) =>
          IpRanges.some(
            ({ CidrIp }) =>
              CidrIp.startsWith(state.myIp) || CidrIp === "0.0.0.0/0",
          ),
      )
      .filter(({ IpProtocol }) => IpProtocol === "tcp")
    },
  ),

```

```

        .filter(({ FromPort }) => FromPort === 80);

        state.myIpRules = myIpRules;
    },
),
new ScenarioOutput(
    "verifiedInboundPort",
    /**
     * @param {{ myIpRules: any[] }} state
     */
    (state) => {
        if (state.myIpRules.length > 0) {
            return MESSAGES.foundIpRules.replace(
                "${IP_RULES}",
                JSON.stringify(state.myIpRules, null, 2),
            );
        } else {
            return MESSAGES.noIpRules;
        }
    },
),
new ScenarioInput(
    "shouldAddInboundRule",
    /**
     * @param {{ myIpRules: any[] }} state
     */
    (state) => {
        if (state.myIpRules.length > 0) {
            return false;
        } else {
            return MESSAGES.noIpRules;
        }
    },
    { type: "confirm" },
),
new ScenarioAction(
    "addInboundRule",
    /**
     * @param {{ defaultSecurityGroup: import('@aws-sdk/client-ec2').SecurityGroup }} state
     */
    async (state) => {
        if (!state.shouldAddInboundRule) {
            return;
        }
    }
);

```

```
    }

    const client = new EC2Client({});
    await client.send(
      new AuthorizeSecurityGroupIngressCommand({
        GroupId: state.defaultSecurityGroup.GroupId,
        CidrIp: `${state.myIp}/32`,
        FromPort: 80,
        ToPort: 80,
        IpProtocol: "tcp",
      }),
    );
  },
),
new ScenarioOutput("addedInboundRule", (state) => {
  if (state.shouldAddInboundRule) {
    return MESSAGES.addedInboundRule.replace("${IP_ADDRESS}", state.myIp);
  } else {
    return false;
  }
}),
new ScenarioOutput("verifyingEndpoint", (state) =>
  MESSAGES.verifyingEndpoint.replace("${DNS_NAME}", state.loadBalancerDns),
),
new ScenarioAction("verifyEndpoint", async (state) => {
  try {
    const response = await retry({ intervalInMs: 2000, maxRetries: 30 }, () =>
      axios.get(`http://${state.loadBalancerDns}`),
    );
    state.endpointResponse = JSON.stringify(response.data, null, 2);
  } catch (e) {
    state.verifyEndpointError = e;
  }
}),
new ScenarioOutput("verifiedEndpoint", (state) => {
  if (state.verifyEndpointError) {
    console.error(state.verifyEndpointError);
  } else {
    return MESSAGES.verifiedEndpoint.replace(
      "${ENDPOINT_RESPONSE}",
      state.endpointResponse,
    );
  }
}),
}),
```

```
    saveState,  
  ];
```

Create steps to run the demo.

```
import { readFileSync } from "node:fs";  
import { join } from "node:path";  
  
import axios from "axios";  
  
import {  
  DescribeTargetGroupsCommand,  
  DescribeTargetHealthCommand,  
  ElasticLoadBalancingV2Client,  
} from "@aws-sdk/client-elastic-load-balancing-v2";  
import {  
  DescribeInstanceInformationCommand,  
  PutParameterCommand,  
  SSMClient,  
  SendCommandCommand,  
} from "@aws-sdk/client-ssm";  
import {  
  IAMClient,  
  CreatePolicyCommand,  
  CreateRoleCommand,  
  AttachRolePolicyCommand,  
  CreateInstanceProfileCommand,  
  AddRoleToInstanceProfileCommand,  
  waitUntilInstanceProfileExists,  
} from "@aws-sdk/client-iam";  
import {  
  AutoScalingClient,  
  DescribeAutoScalingGroupsCommand,  
  TerminateInstanceInAutoScalingGroupCommand,  
} from "@aws-sdk/client-auto-scaling";  
import {  
  DescribeIamInstanceProfileAssociationsCommand,  
  EC2Client,  
  RebootInstancesCommand,  
  ReplaceIamInstanceProfileAssociationCommand,  
} from "@aws-sdk/client-ec2";
```



```
import {
  ScenarioAction,
  ScenarioInput,
  ScenarioOutput,
} from "@aws-doc-sdk-examples/lib/scenario/scenario.js";
import { retry } from "@aws-doc-sdk-examples/lib/utils/util-timers.js";

import { MESSAGES, NAMES, RESOURCES_PATH } from "./constants.js";
import { findLoadBalancer } from "./shared.js";

const getRecommendation = new ScenarioAction(
  "getRecommendation",
  async (state) => {
    const loadBalancer = await findLoadBalancer(NAMES.loadBalancerName);
    if (loadBalancer) {
      state.loadBalancerDnsName = loadBalancer.DNSName;
      try {
        state.recommendation = (
          await axios.get(`http://${state.loadBalancerDnsName}`)
        ).data;
      } catch (e) {
        state.recommendation = e instanceof Error ? e.message : e;
      }
    } else {
      throw new Error(MESSAGES.demoFindLoadBalancerError);
    }
  },
);

const getRecommendationResult = new ScenarioOutput(
  "getRecommendationResult",
  (state) =>
    `Recommendation:\n${JSON.stringify(state.recommendation, null, 2)}`,
  { preformatted: true },
);

const getHealthCheck = new ScenarioAction("getHealthCheck", async (state) => {
  const client = new ElasticLoadBalancingV2Client({});
  const { TargetGroups } = await client.send(
    new DescribeTargetGroupsCommand({
      Names: [NAMES.loadBalancerTargetGroupName],
    }),
  );
});
```

```

const { TargetHealthDescriptions } = await client.send(
  new DescribeTargetHealthCommand({
    TargetGroupArn: TargetGroups[0].TargetGroupArn,
  }),
);
state.targetHealthDescriptions = TargetHealthDescriptions;
});

const getHealthCheckResult = new ScenarioOutput(
  "getHealthCheckResult",
  /**
   * @param {{ targetHealthDescriptions: import('@aws-sdk/client-elastic-load-
balancing-v2').TargetHealthDescription[] }} state
   */
  (state) => {
    const status = state.targetHealthDescriptions
      .map((th) => `${th.Target.Id}: ${th.TargetHealth.State}`)
      .join("\n");
    return `Health check:\n${status}`;
  },
  { preformatted: true },
);

const loadBalancerLoop = new ScenarioAction(
  "loadBalancerLoop",
  getRecommendation.action,
  {
    whileConfig: {
      whileFn: ({ loadBalancerCheck }) => loadBalancerCheck,
      input: new ScenarioInput(
        "loadBalancerCheck",
        MESSAGES.demoLoadBalancerCheck,
        {
          type: "confirm",
        },
      ),
      output: getRecommendationResult,
    },
  },
);

const healthCheckLoop = new ScenarioAction(
  "healthCheckLoop",
  getHealthCheck.action,

```

```
{
  whileConfig: {
    whileFn: ({ healthCheck }) => healthCheck,
    input: new ScenarioInput("healthCheck", MESSAGES.demoHealthCheck, {
      type: "confirm",
    }),
    output: getHealthCheckResult,
  },
},
);

const statusSteps = [
  getRecommendation,
  getRecommendationResult,
  getHealthCheck,
  getHealthCheckResult,
];

/**
 * @type {import('@aws-doc-sdk-examples/lib/scenario.js').Step[]}
 */
export const demoSteps = [
  new ScenarioOutput("header", MESSAGES.demoHeader, { header: true }),
  new ScenarioOutput("sanityCheck", MESSAGES.demoSanityCheck),
  ...statusSteps,
  new ScenarioInput(
    "brokenDependencyConfirmation",
    MESSAGES.demoBrokenDependencyConfirmation,
    { type: "confirm" },
  ),
  new ScenarioAction("brokenDependency", async (state) => {
    if (!state.brokenDependencyConfirmation) {
      process.exit();
    } else {
      const client = new SSMClient({});
      state.badTableName = `fake-table-${Date.now()}`;
      await client.send(
        new PutParameterCommand({
          Name: NAMES.ssmTableNameKey,
          Value: state.badTableName,
          Overwrite: true,
          Type: "String",
        })
      ),
    }
  });
];
```

```
    }
  }},
  new ScenarioOutput("testBrokenDependency", (state) =>
    MESSAGES.demoTestBrokenDependency.replace(
      "${TABLE_NAME}",
      state.badTableName,
    ),
  ),
  ...statusSteps,
  new ScenarioInput(
    "staticResponseConfirmation",
    MESSAGES.demoStaticResponseConfirmation,
    { type: "confirm" },
  ),
  new ScenarioAction("staticResponse", async (state) => {
    if (!state.staticResponseConfirmation) {
      process.exit();
    } else {
      const client = new SSMClient({});
      await client.send(
        new PutParameterCommand({
          Name: NAMES.ssmFailureResponseKey,
          Value: "static",
          Overwrite: true,
          Type: "String",
        })),
      );
    }
  }},
  new ScenarioOutput("testStaticResponse", MESSAGES.demoTestStaticResponse),
  ...statusSteps,
  new ScenarioInput(
    "badCredentialsConfirmation",
    MESSAGES.demoBadCredentialsConfirmation,
    { type: "confirm" },
  ),
  new ScenarioAction("badCredentialsExit", (state) => {
    if (!state.badCredentialsConfirmation) {
      process.exit();
    }
  }},
  new ScenarioAction("fixDynamoDBName", async () => {
    const client = new SSMClient({});
    await client.send(
```

```
    new PutParameterCommand({
      Name: NAMES.ssmTableNameKey,
      Value: NAMES.tableName,
      Overwrite: true,
      Type: "String",
    }),
  );
}),
new ScenarioAction(
  "badCredentials",
  /**
   * @param {{ targetInstance: import('@aws-sdk/client-auto-scaling').Instance }}
state
  */
  async (state) => {
    await createSsmOnlyInstanceProfile();
    const autoScalingClient = new AutoScalingClient({});
    const { AutoScalingGroups } = await autoScalingClient.send(
      new DescribeAutoScalingGroupsCommand({
        AutoScalingGroupNames: [NAMES.autoScalingGroupName],
      }),
    );
    state.targetInstance = AutoScalingGroups[0].Instances[0];
    const ec2Client = new EC2Client({});
    const { IamInstanceProfileAssociations } = await ec2Client.send(
      new DescribeIamInstanceProfileAssociationsCommand({
        Filters: [
          { Name: "instance-id", Values: [state.targetInstance.InstanceId] },
        ],
      }),
    );
    state.instanceProfileAssociationId =
      IamInstanceProfileAssociations[0].AssociationId;
    await retry({ intervalInMs: 1000, maxRetries: 30 }, () =>
      ec2Client.send(
        new ReplaceIamInstanceProfileAssociationCommand({
          AssociationId: state.instanceProfileAssociationId,
          IamInstanceProfile: { Name: NAMES.ssmOnlyInstanceProfileName },
        }),
      ),
    );

    await ec2Client.send(
      new RebootInstancesCommand({
```

```
        InstanceIds: [state.targetInstance.InstanceId],
    })),
  );

const ssmClient = new SSMClient({});
await retry({ intervalInMs: 20000, maxRetries: 15 }, async () => {
  const { InstanceInformationList } = await ssmClient.send(
    new DescribeInstanceInformationCommand({}),
  );

  const instance = InstanceInformationList.find(
    (info) => info.InstanceId === state.targetInstance.InstanceId,
  );

  if (!instance) {
    throw new Error("Instance not found.");
  }
});

await ssmClient.send(
  new SendCommandCommand({
    InstanceIds: [state.targetInstance.InstanceId],
    DocumentName: "AWS-RunShellScript",
    Parameters: { commands: ["cd / && sudo python3 server.py 80"] },
  }),
);
},
),
new ScenarioOutput(
  "testBadCredentials",
  /**
   * @param {{ targetInstance: import('@aws-sdk/client-ssm').InstanceInformation}}
state
  */
  (state) =>
    MESSAGES.demoTestBadCredentials.replace(
      "${INSTANCE_ID}",
      state.targetInstance.InstanceId,
    ),
),
loadBalancerLoop,
new ScenarioInput(
  "deepHealthCheckConfirmation",
  MESSAGES.demoDeepHealthCheckConfirmation,
```

```

    { type: "confirm" },
  ),
  new ScenarioAction("deepHealthCheckExit", (state) => {
    if (!state.deepHealthCheckConfirmation) {
      process.exit();
    }
  }),
  new ScenarioAction("deepHealthCheck", async () => {
    const client = new SSMClient({});
    await client.send(
      new PutParameterCommand({
        Name: NAMES.ssmHealthCheckKey,
        Value: "deep",
        Overwrite: true,
        Type: "String",
      })
    );
  }),
  new ScenarioOutput("testDeepHealthCheck", MESSAGES.demoTestDeepHealthCheck),
  healthCheckLoop,
  loadBalancerLoop,
  new ScenarioInput(
    "killInstanceConfirmation",
    /**
     * @param {{ targetInstance: import('@aws-sdk/client-
    ssm').InstanceInformation }} state
     */
    (state) =>
      MESSAGES.demoKillInstanceConfirmation.replace(
        "${INSTANCE_ID}",
        state.targetInstance.InstanceId,
      ),
    { type: "confirm" },
  ),
  new ScenarioAction("killInstanceExit", (state) => {
    if (!state.killInstanceConfirmation) {
      process.exit();
    }
  }),
  new ScenarioAction(
    "killInstance",
    /**
     * @param {{ targetInstance: import('@aws-sdk/client-
    ssm').InstanceInformation }} state

```

```
    */
    async (state) => {
      const client = new AutoScalingClient({});
      await client.send(
        new TerminateInstanceInAutoScalingGroupCommand({
          InstanceId: state.targetInstance.InstanceId,
          ShouldDecrementDesiredCapacity: false,
        }),
      );
    },
  ),
  new ScenarioOutput("testKillInstance", MESSAGES.demoTestKillInstance),
  healthCheckLoop,
  loadBalancerLoop,
  new ScenarioInput("failOpenConfirmation", MESSAGES.demoFailOpenConfirmation, {
    type: "confirm",
  }),
  new ScenarioAction("failOpenExit", (state) => {
    if (!state.failOpenConfirmation) {
      process.exit();
    }
  }),
  new ScenarioAction("failOpen", () => {
    const client = new SSMClient({});
    return client.send(
      new PutParameterCommand({
        Name: NAMES.ssmTableNameKey,
        Value: `fake-table-${Date.now()}`,
        Overwrite: true,
        Type: "String",
      }),
    );
  }),
  new ScenarioOutput("testFailOpen", MESSAGES.demoFailOpenTest),
  healthCheckLoop,
  loadBalancerLoop,
  new ScenarioInput(
    "resetTableConfirmation",
    MESSAGES.demoResetTableConfirmation,
    { type: "confirm" },
  ),
  new ScenarioAction("resetTableExit", (state) => {
    if (!state.resetTableConfirmation) {
      process.exit();
    }
  })
}
```



```
    }
  }),
  new ScenarioAction("resetTable", async () => {
    const client = new SSMClient({});
    await client.send(
      new PutParameterCommand({
        Name: NAMES.ssmTableNameKey,
        Value: NAMES.tableName,
        Overwrite: true,
        Type: "String",
      }),
    );
  }),
  new ScenarioOutput("testResetTable", MESSAGES.demoTestResetTable),
  healthCheckLoop,
  loadBalancerLoop,
];

async function createSsmOnlyInstanceProfile() {
  const iamClient = new IAMClient({});
  const { Policy } = await iamClient.send(
    new CreatePolicyCommand({
      PolicyName: NAMES.ssmOnlyPolicyName,
      PolicyDocument: readFileSync(
        join(RESOURCES_PATH, "ssm_only_policy.json"),
      ),
    }),
  );
  await iamClient.send(
    new CreateRoleCommand({
      RoleName: NAMES.ssmOnlyRoleName,
      AssumeRolePolicyDocument: JSON.stringify({
        Version: "2012-10-17",
        Statement: [
          {
            Effect: "Allow",
            Principal: { Service: "ec2.amazonaws.com" },
            Action: "sts:AssumeRole",
          },
        ],
      }),
    ),
  );
  await iamClient.send(
```

```

    new AttachRolePolicyCommand({
      RoleName: NAMES.ssmOnlyRoleName,
      PolicyArn: Policy.Arn,
    }),
  );
  await iamClient.send(
    new AttachRolePolicyCommand({
      RoleName: NAMES.ssmOnlyRoleName,
      PolicyArn: "arn:aws:iam::aws:policy/AmazonSSMManagedInstanceCore",
    }),
  );
  const { InstanceProfile } = await iamClient.send(
    new CreateInstanceProfileCommand({
      InstanceProfileName: NAMES.ssmOnlyInstanceProfileName,
    }),
  );
  await waitUntilInstanceProfileExists(
    { client: iamClient },
    { InstanceProfileName: NAMES.ssmOnlyInstanceProfileName },
  );
  await iamClient.send(
    new AddRoleToInstanceProfileCommand({
      InstanceProfileName: NAMES.ssmOnlyInstanceProfileName,
      RoleName: NAMES.ssmOnlyRoleName,
    }),
  );

  return InstanceProfile;
}

```

Create steps to destroy all of the resources.

```

import { unlinkSync } from "node:fs";

import { DynamoDBClient, DeleteTableCommand } from "@aws-sdk/client-dynamodb";
import {
  EC2Client,
  DeleteKeyPairCommand,
  DeleteLaunchTemplateCommand,
  RevokeSecurityGroupIngressCommand,
} from "@aws-sdk/client-ec2";
import {

```

```

    IAMClient,
    DeleteInstanceProfileCommand,
    RemoveRoleFromInstanceProfileCommand,
    DeletePolicyCommand,
    DeleteRoleCommand,
    DetachRolePolicyCommand,
    paginateListPolicies,
} from "@aws-sdk/client-iam";
import {
    AutoScalingClient,
    DeleteAutoScalingGroupCommand,
    TerminateInstanceInAutoScalingGroupCommand,
    UpdateAutoScalingGroupCommand,
    paginateDescribeAutoScalingGroups,
} from "@aws-sdk/client-auto-scaling";
import {
    DeleteLoadBalancerCommand,
    DeleteTargetGroupCommand,
    DescribeTargetGroupsCommand,
    ElasticLoadBalancingV2Client,
} from "@aws-sdk/client-elastic-load-balancing-v2";

import {
    ScenarioOutput,
    ScenarioInput,
    ScenarioAction,
} from "@aws-doc-sdk-examples/lib/scenario/index.js";
import { loadState } from "@aws-doc-sdk-examples/lib/scenario/steps-common.js";
import { retry } from "@aws-doc-sdk-examples/lib/utils/util-timers.js";

import { MESSAGES, NAMES } from "./constants.js";
import { findLoadBalancer } from "./shared.js";

/**
 * @type {import('@aws-doc-sdk-examples/lib/scenario.js').Step[]}
 */
export const destroySteps = [
    loadState,
    new ScenarioInput("destroy", MESSAGES.destroy, { type: "confirm" }),
    new ScenarioAction(
        "abort",
        (state) => state.destroy === false && process.exit(),
    ),
    new ScenarioAction("deleteTable", async (c) => {

```

```
    try {
      const client = new DynamoDBClient({});
      await client.send(new DeleteTableCommand({ TableName: NAMES.tableName }));
    } catch (e) {
      c.deleteTableError = e;
    }
  })),
  new ScenarioOutput("deleteTableResult", (state) => {
    if (state.deleteTableError) {
      console.error(state.deleteTableError);
      return MESSAGES.deleteTableError.replace(
        "${TABLE_NAME}",
        NAMES.tableName,
      );
    } else {
      return MESSAGES.deletedTable.replace("${TABLE_NAME}", NAMES.tableName);
    }
  })),
  new ScenarioAction("deleteKeyPair", async (state) => {
    try {
      const client = new EC2Client({});
      await client.send(
        new DeleteKeyPairCommand({ KeyName: NAMES.keyPairName }),
      );
      unlinkSync(`${NAMES.keyPairName}.pem`);
    } catch (e) {
      state.deleteKeyPairError = e;
    }
  })),
  new ScenarioOutput("deleteKeyPairResult", (state) => {
    if (state.deleteKeyPairError) {
      console.error(state.deleteKeyPairError);
      return MESSAGES.deleteKeyPairError.replace(
        "${KEY_PAIR_NAME}",
        NAMES.keyPairName,
      );
    } else {
      return MESSAGES.deletedKeyPair.replace(
        "${KEY_PAIR_NAME}",
        NAMES.keyPairName,
      );
    }
  })),
  new ScenarioAction("detachPolicyFromRole", async (state) => {
```

```
try {
  const client = new IAMClient({});
  const policy = await findPolicy(NAMES.instancePolicyName);

  if (!policy) {
    state.detachPolicyFromRoleError = new Error(
      `Policy ${NAMES.instancePolicyName} not found.`
    );
  } else {
    await client.send(
      new DetachRolePolicyCommand({
        RoleName: NAMES.instanceRoleName,
        PolicyArn: policy.Arn,
      }),
    );
  }
} catch (e) {
  state.detachPolicyFromRoleError = e;
}
}),
new ScenarioOutput("detachedPolicyFromRole", (state) => {
  if (state.detachPolicyFromRoleError) {
    console.error(state.detachPolicyFromRoleError);
    return MESSAGES.detachPolicyFromRoleError
      .replace("${INSTANCE_POLICY_NAME}", NAMES.instancePolicyName)
      .replace("${INSTANCE_ROLE_NAME}", NAMES.instanceRoleName);
  } else {
    return MESSAGES.detachedPolicyFromRole
      .replace("${INSTANCE_POLICY_NAME}", NAMES.instancePolicyName)
      .replace("${INSTANCE_ROLE_NAME}", NAMES.instanceRoleName);
  }
}),
new ScenarioAction("deleteInstancePolicy", async (state) => {
  const client = new IAMClient({});
  const policy = await findPolicy(NAMES.instancePolicyName);

  if (!policy) {
    state.deletePolicyError = new Error(
      `Policy ${NAMES.instancePolicyName} not found.`
    );
  } else {
    return client.send(
      new DeletePolicyCommand({
        PolicyArn: policy.Arn,
```

```
    }),
  );
}
}),
new ScenarioOutput("deletePolicyResult", (state) => {
  if (state.deletePolicyError) {
    console.error(state.deletePolicyError);
    return MESSAGES.deletePolicyError.replace(
      "${INSTANCE_POLICY_NAME}",
      NAMES.instancePolicyName,
    );
  } else {
    return MESSAGES.deletedPolicy.replace(
      "${INSTANCE_POLICY_NAME}",
      NAMES.instancePolicyName,
    );
  }
}),
new ScenarioAction("removeRoleFromInstanceProfile", async (state) => {
  try {
    const client = new IAMClient({});
    await client.send(
      new RemoveRoleFromInstanceProfileCommand({
        RoleName: NAMES.instanceRoleName,
        InstanceProfileName: NAMES.instanceProfileName,
      }),
    );
  } catch (e) {
    state.removeRoleFromInstanceProfileError = e;
  }
}),
new ScenarioOutput("removeRoleFromInstanceProfileResult", (state) => {
  if (state.removeRoleFromInstanceProfile) {
    console.error(state.removeRoleFromInstanceProfileError);
    return MESSAGES.removeRoleFromInstanceProfileError
      .replace("${INSTANCE_PROFILE_NAME}", NAMES.instanceProfileName)
      .replace("${INSTANCE_ROLE_NAME}", NAMES.instanceRoleName);
  } else {
    return MESSAGES.removedRoleFromInstanceProfile
      .replace("${INSTANCE_PROFILE_NAME}", NAMES.instanceProfileName)
      .replace("${INSTANCE_ROLE_NAME}", NAMES.instanceRoleName);
  }
}),
new ScenarioAction("deleteInstanceRole", async (state) => {
```

```
    try {
      const client = new IAMClient({});
      await client.send(
        new DeleteRoleCommand({
          RoleName: NAMES.instanceRoleName,
        }),
      );
    } catch (e) {
      state.deleteInstanceRoleError = e;
    }
  })),
  new ScenarioOutput("deleteInstanceRoleResult", (state) => {
    if (state.deleteInstanceRoleError) {
      console.error(state.deleteInstanceRoleError);
      return MESSAGES.deleteInstanceRoleError.replace(
        "${INSTANCE_ROLE_NAME}",
        NAMES.instanceRoleName,
      );
    } else {
      return MESSAGES.deletedInstanceRole.replace(
        "${INSTANCE_ROLE_NAME}",
        NAMES.instanceRoleName,
      );
    }
  })),
  new ScenarioAction("deleteInstanceProfile", async (state) => {
    try {
      const client = new IAMClient({});
      await client.send(
        new DeleteInstanceProfileCommand({
          InstanceProfileName: NAMES.instanceProfileName,
        }),
      );
    } catch (e) {
      state.deleteInstanceProfileError = e;
    }
  })),
  new ScenarioOutput("deleteInstanceProfileResult", (state) => {
    if (state.deleteInstanceProfileError) {
      console.error(state.deleteInstanceProfileError);
      return MESSAGES.deleteInstanceProfileError.replace(
        "${INSTANCE_PROFILE_NAME}",
        NAMES.instanceProfileName,
      );
    }
  }));
```

```
    } else {
      return MESSAGES.deletedInstanceProfile.replace(
        "${INSTANCE_PROFILE_NAME}",
        NAMES.instanceProfileName,
      );
    }
  })),
  new ScenarioAction("deleteAutoScalingGroup", async (state) => {
    try {
      await terminateGroupInstances(NAMES.autoScalingGroupName);
      await retry({ intervalInMs: 60000, maxRetries: 60 }, async () => {
        await deleteAutoScalingGroup(NAMES.autoScalingGroupName);
      });
    } catch (e) {
      state.deleteAutoScalingGroupError = e;
    }
  })),
  new ScenarioOutput("deleteAutoScalingGroupResult", (state) => {
    if (state.deleteAutoScalingGroupError) {
      console.error(state.deleteAutoScalingGroupError);
      return MESSAGES.deleteAutoScalingGroupError.replace(
        "${AUTO_SCALING_GROUP_NAME}",
        NAMES.autoScalingGroupName,
      );
    } else {
      return MESSAGES.deletedAutoScalingGroup.replace(
        "${AUTO_SCALING_GROUP_NAME}",
        NAMES.autoScalingGroupName,
      );
    }
  })),
  new ScenarioAction("deleteLaunchTemplate", async (state) => {
    const client = new EC2Client({});
    try {
      await client.send(
        new DeleteLaunchTemplateCommand({
          LaunchTemplateName: NAMES.launchTemplateName,
        }),
      );
    } catch (e) {
      state.deleteLaunchTemplateError = e;
    }
  })),
  new ScenarioOutput("deleteLaunchTemplateResult", (state) => {
```



```
    if (state.deleteLaunchTemplateError) {
      console.error(state.deleteLaunchTemplateError);
      return MESSAGES.deleteLaunchTemplateError.replace(
        "${LAUNCH_TEMPLATE_NAME}",
        NAMES.launchTemplateName,
      );
    } else {
      return MESSAGES.deletedLaunchTemplate.replace(
        "${LAUNCH_TEMPLATE_NAME}",
        NAMES.launchTemplateName,
      );
    }
  })),
  new ScenarioAction("deleteLoadBalancer", async (state) => {
    try {
      const client = new ElasticLoadBalancingV2Client({});
      const loadBalancer = await findLoadBalancer(NAMES.loadBalancerName);
      await client.send(
        new DeleteLoadBalancerCommand({
          LoadBalancerArn: loadBalancer.LoadBalancerArn,
        })),
      );
      await retry({ intervalInMs: 1000, maxRetries: 60 }, async () => {
        const lb = await findLoadBalancer(NAMES.loadBalancerName);
        if (lb) {
          throw new Error("Load balancer still exists.");
        }
      });
    } catch (e) {
      state.deleteLoadBalancerError = e;
    }
  })),
  new ScenarioOutput("deleteLoadBalancerResult", (state) => {
    if (state.deleteLoadBalancerError) {
      console.error(state.deleteLoadBalancerError);
      return MESSAGES.deleteLoadBalancerError.replace(
        "${LB_NAME}",
        NAMES.loadBalancerName,
      );
    } else {
      return MESSAGES.deletedLoadBalancer.replace(
        "${LB_NAME}",
        NAMES.loadBalancerName,
      );
    }
  });
}
```

```
    }
  }},
  new ScenarioAction("deleteLoadBalancerTargetGroup", async (state) => {
    const client = new ElasticLoadBalancingV2Client({});
    try {
      const { TargetGroups } = await client.send(
        new DescribeTargetGroupsCommand({
          Names: [NAMES.loadBalancerTargetGroupName],
        }),
      );
      await retry({ intervalInMs: 1000, maxRetries: 30 }, () =>
        client.send(
          new DeleteTargetGroupCommand({
            TargetGroupArn: TargetGroups[0].TargetGroupArn,
          }),
        );
    } catch (e) {
      state.deleteLoadBalancerTargetGroupError = e;
    }
  }},
  new ScenarioOutput("deleteLoadBalancerTargetGroupResult", (state) => {
    if (state.deleteLoadBalancerTargetGroupError) {
      console.error(state.deleteLoadBalancerTargetGroupError);
      return MESSAGES.deleteLoadBalancerTargetGroupError.replace(
        "${TARGET_GROUP_NAME}",
        NAMES.loadBalancerTargetGroupName,
      );
    } else {
      return MESSAGES.deletedLoadBalancerTargetGroup.replace(
        "${TARGET_GROUP_NAME}",
        NAMES.loadBalancerTargetGroupName,
      );
    }
  }},
  new ScenarioAction("detachSsmOnlyRoleFromProfile", async (state) => {
    try {
      const client = new IAMClient({});
      await client.send(
        new RemoveRoleFromInstanceProfileCommand({
          InstanceProfileName: NAMES.ssmOnlyInstanceProfileName,
          RoleName: NAMES.ssmOnlyRoleName,
        }),
      );
    }
  }},
```

```
    );
  } catch (e) {
    state.detachSsmOnlyRoleFromProfileError = e;
  }
}),
new ScenarioOutput("detachSsmOnlyRoleFromProfileResult", (state) => {
  if (state.detachSsmOnlyRoleFromProfileError) {
    console.error(state.detachSsmOnlyRoleFromProfileError);
    return MESSAGES.detachSsmOnlyRoleFromProfileError
      .replace("${ROLE_NAME}", NAMES.ssmOnlyRoleName)
      .replace("${PROFILE_NAME}", NAMES.ssmOnlyInstanceProfileName);
  } else {
    return MESSAGES.detachedSsmOnlyRoleFromProfile
      .replace("${ROLE_NAME}", NAMES.ssmOnlyRoleName)
      .replace("${PROFILE_NAME}", NAMES.ssmOnlyInstanceProfileName);
  }
}),
new ScenarioAction("detachSsmOnlyCustomRolePolicy", async (state) => {
  try {
    const iamClient = new IAMClient({});
    const ssmOnlyPolicy = await findPolicy(NAMES.ssmOnlyPolicyName);
    await iamClient.send(
      new DetachRolePolicyCommand({
        RoleName: NAMES.ssmOnlyRoleName,
        PolicyArn: ssmOnlyPolicy.Arn,
      })),
    );
  } catch (e) {
    state.detachSsmOnlyCustomRolePolicyError = e;
  }
}),
new ScenarioOutput("detachSsmOnlyCustomRolePolicyResult", (state) => {
  if (state.detachSsmOnlyCustomRolePolicyError) {
    console.error(state.detachSsmOnlyCustomRolePolicyError);
    return MESSAGES.detachSsmOnlyCustomRolePolicyError
      .replace("${ROLE_NAME}", NAMES.ssmOnlyRoleName)
      .replace("${POLICY_NAME}", NAMES.ssmOnlyPolicyName);
  } else {
    return MESSAGES.detachedSsmOnlyCustomRolePolicy
      .replace("${ROLE_NAME}", NAMES.ssmOnlyRoleName)
      .replace("${POLICY_NAME}", NAMES.ssmOnlyPolicyName);
  }
}),
new ScenarioAction("detachSsmOnlyAWSRolePolicy", async (state) => {
```

```
try {
  const iamClient = new IAMClient({});
  await iamClient.send(
    new DetachRolePolicyCommand({
      RoleName: NAMES.ssmOnlyRoleName,
      PolicyArn: "arn:aws:iam::aws:policy/AmazonSSManagedInstanceCore",
    }),
  );
} catch (e) {
  state.detachSsmOnlyAWSRolePolicyError = e;
}
}),
new ScenarioOutput("detachSsmOnlyAWSRolePolicyResult", (state) => {
  if (state.detachSsmOnlyAWSRolePolicyError) {
    console.error(state.detachSsmOnlyAWSRolePolicyError);
    return MESSAGES.detachSsmOnlyAWSRolePolicyError
      .replace("${ROLE_NAME}", NAMES.ssmOnlyRoleName)
      .replace("${POLICY_NAME}", "AmazonSSManagedInstanceCore");
  } else {
    return MESSAGES.detachedSsmOnlyAWSRolePolicy
      .replace("${ROLE_NAME}", NAMES.ssmOnlyRoleName)
      .replace("${POLICY_NAME}", "AmazonSSManagedInstanceCore");
  }
}),
new ScenarioAction("deleteSsmOnlyInstanceProfile", async (state) => {
  try {
    const iamClient = new IAMClient({});
    await iamClient.send(
      new DeleteInstanceProfileCommand({
        InstanceProfileName: NAMES.ssmOnlyInstanceProfileName,
      }),
    );
  } catch (e) {
    state.deleteSsmOnlyInstanceProfileError = e;
  }
}),
new ScenarioOutput("deleteSsmOnlyInstanceProfileResult", (state) => {
  if (state.deleteSsmOnlyInstanceProfileError) {
    console.error(state.deleteSsmOnlyInstanceProfileError);
    return MESSAGES.deleteSsmOnlyInstanceProfileError.replace(
      "${INSTANCE_PROFILE_NAME}",
      NAMES.ssmOnlyInstanceProfileName,
    );
  } else {
```

```
    return MESSAGES.deletedSsmOnlyInstanceProfile.replace(
      "${INSTANCE_PROFILE_NAME}",
      NAMES.ssmOnlyInstanceProfileName,
    );
  }
}),
new ScenarioAction("deleteSsmOnlyPolicy", async (state) => {
  try {
    const iamClient = new IAMClient({});
    const ssmOnlyPolicy = await findPolicy(NAMES.ssmOnlyPolicyName);
    await iamClient.send(
      new DeletePolicyCommand({
        PolicyArn: ssmOnlyPolicy.Arn,
      }),
    );
  } catch (e) {
    state.deleteSsmOnlyPolicyError = e;
  }
}),
new ScenarioOutput("deleteSsmOnlyPolicyResult", (state) => {
  if (state.deleteSsmOnlyPolicyError) {
    console.error(state.deleteSsmOnlyPolicyError);
    return MESSAGES.deleteSsmOnlyPolicyError.replace(
      "${POLICY_NAME}",
      NAMES.ssmOnlyPolicyName,
    );
  } else {
    return MESSAGES.deletedSsmOnlyPolicy.replace(
      "${POLICY_NAME}",
      NAMES.ssmOnlyPolicyName,
    );
  }
}),
new ScenarioAction("deleteSsmOnlyRole", async (state) => {
  try {
    const iamClient = new IAMClient({});
    await iamClient.send(
      new DeleteRoleCommand({
        RoleName: NAMES.ssmOnlyRoleName,
      }),
    );
  } catch (e) {
    state.deleteSsmOnlyRoleError = e;
  }
})
```

```
    }),
    new ScenarioOutput("deleteSsmOnlyRoleResult", (state) => {
      if (state.deleteSsmOnlyRoleError) {
        console.error(state.deleteSsmOnlyRoleError);
        return MESSAGES.deleteSsmOnlyRoleError.replace(
          "${ROLE_NAME}",
          NAMES.ssmOnlyRoleName,
        );
      } else {
        return MESSAGES.deletedSsmOnlyRole.replace(
          "${ROLE_NAME}",
          NAMES.ssmOnlyRoleName,
        );
      }
    })),
    new ScenarioAction(
      "revokeSecurityGroupIngress",
      async (
        /** @type {{ myIp: string, defaultSecurityGroup: { GroupId: string } }} */
state,
      ) => {
        const ec2Client = new EC2Client({});

        try {
          await ec2Client.send(
            new RevokeSecurityGroupIngressCommand({
              GroupId: state.defaultSecurityGroup.GroupId,
              CidrIp: `${state.myIp}/32`,
              FromPort: 80,
              ToPort: 80,
              IpProtocol: "tcp",
            }),
          );
        } catch (e) {
          state.revokeSecurityGroupIngressError = e;
        }
      },
    ),
    new ScenarioOutput("revokeSecurityGroupIngressResult", (state) => {
      if (state.revokeSecurityGroupIngressError) {
        console.error(state.revokeSecurityGroupIngressError);
        return MESSAGES.revokeSecurityGroupIngressError.replace(
          "${IP}",
          state.myIp,
        );
      }
    })),
  ],
);
```

```
    );
  } else {
    return MESSAGES.revokedSecurityGroupIngress.replace("${IP}", state.myIp);
  }
  })),
];

/**
 * @param {string} policyName
 */
async function findPolicy(policyName) {
  const client = new IAMClient({});
  const paginatedPolicies = paginateListPolicies({ client }, {});
  for await (const page of paginatedPolicies) {
    const policy = page.Policies.find((p) => p.PolicyName === policyName);
    if (policy) {
      return policy;
    }
  }
}

/**
 * @param {string} groupName
 */
async function deleteAutoScalingGroup(groupName) {
  const client = new AutoScalingClient({});
  try {
    await client.send(
      new DeleteAutoScalingGroupCommand({
        AutoScalingGroupName: groupName,
      }),
    );
  } catch (err) {
    if (!(err instanceof Error)) {
      throw err;
    } else {
      console.log(err.name);
      throw err;
    }
  }
}

/**
 * @param {string} groupName
```

```
*/
async function terminateGroupInstances(groupName) {
  const autoScalingClient = new AutoScalingClient({});
  const group = await findAutoScalingGroup(groupName);
  await autoScalingClient.send(
    new UpdateAutoScalingGroupCommand({
      AutoScalingGroupName: group.AutoScalingGroupName,
      MinSize: 0,
    }),
  );
  for (const i of group.Instances) {
    await retry({ intervalInMs: 1000, maxRetries: 30 }, () =>
      autoScalingClient.send(
        new TerminateInstanceInAutoScalingGroupCommand({
          InstanceId: i.InstanceId,
          ShouldDecrementDesiredCapacity: true,
        }),
      ),
    );
  }
}

async function findAutoScalingGroup(groupName) {
  const client = new AutoScalingClient({});
  const paginatedGroups = paginateDescribeAutoScalingGroups({ client }, {});
  for await (const page of paginatedGroups) {
    const group = page.AutoScalingGroups.find(
      (g) => g.AutoScalingGroupName === groupName,
    );
    if (group) {
      return group;
    }
  }
  throw new Error(`Auto scaling group ${groupName} not found.`);
}
```

- For API details, see the following topics in *AWS SDK for JavaScript API Reference*.
 - [AttachLoadBalancerTargetGroups](#)
 - [CreateAutoScalingGroup](#)
 - [CreateInstanceProfile](#)
 - [CreateLaunchTemplate](#)

- [CreateListener](#)
- [CreateLoadBalancer](#)
- [CreateTargetGroup](#)
- [DeleteAutoScalingGroup](#)
- [DeleteInstanceProfile](#)
- [DeleteLaunchTemplate](#)
- [DeleteLoadBalancer](#)
- [DeleteTargetGroup](#)
- [DescribeAutoScalingGroups](#)
- [DescribeAvailabilityZones](#)
- [DescribeIamInstanceProfileAssociations](#)
- [DescribeInstances](#)
- [DescribeLoadBalancers](#)
- [DescribeSubnets](#)
- [DescribeTargetGroups](#)
- [DescribeTargetHealth](#)
- [DescribeVpcs](#)
- [RebootInstances](#)
- [ReplacesIamInstanceProfileAssociation](#)
- [TerminateInstanceInAutoScalingGroup](#)
- [UpdateAutoScalingGroup](#)

Elastic Load Balancing - Version 2 examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with Elastic Load Balancing - Version 2.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Scenarios are code examples that show you how to accomplish specific tasks by calling multiple functions within a service or combined with other AWS services.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Get started

Hello Elastic Load Balancing

The following code examples show how to get started using Elastic Load Balancing.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import {
  ElasticLoadBalancingV2Client,
  DescribeLoadBalancersCommand,
} from "@aws-sdk/client-elastic-load-balancing-v2";

export async function main() {
  const client = new ElasticLoadBalancingV2Client({});
  const { LoadBalancers } = await client.send(
    new DescribeLoadBalancersCommand({}),
  );
  const loadBalancersList = LoadBalancers.map(
    (lb) => `• ${lb.LoadBalancerName}: ${lb.DNSName}`,
  ).join("\n");
  console.log(
    "Hello, Elastic Load Balancing! Let's list some of your load balancers:\n",
    loadBalancersList,
  );
}

// Call function if run directly
import { fileURLToPath } from "url";
```

```
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  main();
}
```

- For API details, see [DescribeLoadBalancers](#) in *AWS SDK for JavaScript API Reference*.

Topics

- [Actions](#)
- [Scenarios](#)

Actions

CreateListener

The following code example shows how to use `CreateListener`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const client = new ElasticLoadBalancingV2Client({});
const { Listeners } = await client.send(
  new CreateListenerCommand({
    LoadBalancerArn: state.loadBalancerArn,
    Protocol: state.targetGroupProtocol,
    Port: state.targetGroupPort,
    DefaultActions: [
      { Type: "forward", TargetGroupArn: state.targetGroupArn },
    ],
  })),
);
```

- For API details, see [CreateListener](#) in *AWS SDK for JavaScript API Reference*.

CreateLoadBalancer

The following code example shows how to use CreateLoadBalancer.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const client = new ElasticLoadBalancingV2Client({});
const { LoadBalancers } = await client.send(
  new CreateLoadBalancerCommand({
    Name: NAMES.loadBalancerName,
    Subnets: state.subnets,
  }),
);
state.loadBalancerDns = LoadBalancers[0].DNSName;
state.loadBalancerArn = LoadBalancers[0].LoadBalancerArn;
await waitUntilLoadBalancerAvailable(
  { client },
  { Names: [NAMES.loadBalancerName] },
);
```

- For API details, see [CreateLoadBalancer](#) in *AWS SDK for JavaScript API Reference*.

CreateTargetGroup

The following code example shows how to use CreateTargetGroup.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const client = new ElasticLoadBalancingV2Client({});
const { TargetGroups } = await client.send(
  new CreateTargetGroupCommand({
    Name: NAMES.loadBalancerTargetGroupName,
    Protocol: "HTTP",
    Port: 80,
    HealthCheckPath: "/healthcheck",
    HealthCheckIntervalSeconds: 10,
    HealthCheckTimeoutSeconds: 5,
    HealthyThresholdCount: 2,
    UnhealthyThresholdCount: 2,
    VpcId: state.defaultVpc,
  }),
);
```

- For API details, see [CreateTargetGroup](#) in *AWS SDK for JavaScript API Reference*.

DeleteLoadBalancer

The following code example shows how to use DeleteLoadBalancer.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const client = new ElasticLoadBalancingV2Client({});
const loadBalancer = await findLoadBalancer(NAMES.loadBalancerName);
await client.send(
  new DeleteLoadBalancerCommand({
    LoadBalancerArn: loadBalancer.LoadBalancerArn,
  }),
);
await retry({ intervalInMs: 1000, maxRetries: 60 }, async () => {
  const lb = await findLoadBalancer(NAMES.loadBalancerName);
  if (lb) {
    throw new Error("Load balancer still exists.");
  }
});
```

```
    }  
  });
```

- For API details, see [DeleteLoadBalancer](#) in *AWS SDK for JavaScript API Reference*.

DeleteTargetGroup

The following code example shows how to use `DeleteTargetGroup`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const client = new ElasticLoadBalancingV2Client({});  
try {  
  const { TargetGroups } = await client.send(  
    new DescribeTargetGroupsCommand({  
      Names: [NAMES.loadBalancerTargetGroupName],  
    }),  
  );  
  
  await retry({ intervalInMs: 1000, maxRetries: 30 }, () =>  
    client.send(  
      new DeleteTargetGroupCommand({  
        TargetGroupArn: TargetGroups[0].TargetGroupArn,  
      }),  
    ),  
  );  
} catch (e) {  
  state.deleteLoadBalancerTargetGroupError = e;  
}
```

- For API details, see [DeleteTargetGroup](#) in *AWS SDK for JavaScript API Reference*.

DescribeLoadBalancers

The following code example shows how to use DescribeLoadBalancers.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import {
  ElasticLoadBalancingV2Client,
  DescribeLoadBalancersCommand,
} from "@aws-sdk/client-elastic-load-balancing-v2";

export async function main() {
  const client = new ElasticLoadBalancingV2Client({});
  const { LoadBalancers } = await client.send(
    new DescribeLoadBalancersCommand({}),
  );
  const loadBalancersList = LoadBalancers.map(
    (lb) => `• ${lb.LoadBalancerName}: ${lb.DNSName}`,
  ).join("\n");
  console.log(
    "Hello, Elastic Load Balancing! Let's list some of your load balancers:\n",
    loadBalancersList,
  );
}

// Call function if run directly
import { fileURLToPath } from "url";
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  main();
}
```

- For API details, see [DescribeLoadBalancers](#) in *AWS SDK for JavaScript API Reference*.

DescribeTargetGroups

The following code example shows how to use DescribeTargetGroups.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const client = new ElasticLoadBalancingV2Client({});
const { TargetGroups } = await client.send(
  new DescribeTargetGroupsCommand({
    Names: [NAMES.loadBalancerTargetGroupName],
  }),
);
```

- For API details, see [DescribeTargetGroups](#) in *AWS SDK for JavaScript API Reference*.

DescribeTargetHealth

The following code example shows how to use DescribeTargetHealth.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const { TargetHealthDescriptions } = await client.send(
  new DescribeTargetHealthCommand({
    TargetGroupArn: TargetGroups[0].TargetGroupArn,
  }),
);
```


- For API details, see [DescribeTargetHealth](#) in *AWS SDK for JavaScript API Reference*.

Scenarios

Build and manage a resilient service

The following code example shows how to create a load-balanced web service that returns book, movie, and song recommendations. The example shows how the service responds to failures, and how to restructure the service for more resilience when failures occur.

- Use an Amazon EC2 Auto Scaling group to create Amazon Elastic Compute Cloud (Amazon EC2) instances based on a launch template and to keep the number of instances in a specified range.
- Handle and distribute HTTP requests with Elastic Load Balancing.
- Monitor the health of instances in an Auto Scaling group and forward requests only to healthy instances.
- Run a Python web server on each EC2 instance to handle HTTP requests. The web server responds with recommendations and health checks.
- Simulate a recommendation service with an Amazon DynamoDB table.
- Control web server response to requests and health checks by updating AWS Systems Manager parameters.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Run the interactive scenario at a command prompt.

```
#!/usr/bin/env node
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

import {
  Scenario,
```

```
    parseScenarioArgs,
  } from "@aws-doc-sdk-examples/lib/scenario/index.js";

/**
 * The workflow steps are split into three stages:
 * - deploy
 * - demo
 * - destroy
 *
 * Each of these stages has a corresponding file prefixed with steps-*.
 */
import { deploySteps } from "./steps-deploy.js";
import { demoSteps } from "./steps-demo.js";
import { destroySteps } from "./steps-destroy.js";

/**
 * The context is passed to every scenario. Scenario steps
 * will modify the context.
 */
const context = {};

/**
 * Three Scenarios are created for the workflow. A Scenario is an orchestration
 class
 * that simplifies running a series of steps.
 */
export const scenarios = {
  // Deploys all resources necessary for the workflow.
  deploy: new Scenario("Resilient Workflow - Deploy", deploySteps, context),
  // Demonstrates how a fragile web service can be made more resilient.
  demo: new Scenario("Resilient Workflow - Demo", demoSteps, context),
  // Destroys the resources created for the workflow.
  destroy: new Scenario("Resilient Workflow - Destroy", destroySteps, context),
};

// Call function if run directly
import { fileURLToPath } from "url";

if (process.argv[1] === fileURLToPath(import.meta.url)) {
  parseScenarioArgs(scenarios);
}
```

Create steps to deploy all of the resources.

```
import { join } from "node:path";
import { readFileSync, writeFileSync } from "node:fs";
import axios from "axios";

import {
  BatchWriteItemCommand,
  CreateTableCommand,
  DynamoDBClient,
  waitUntilTableExists,
} from "@aws-sdk/client-dynamodb";
import {
  EC2Client,
  CreateKeyPairCommand,
  CreateLaunchTemplateCommand,
  DescribeAvailabilityZonesCommand,
  DescribeVpcsCommand,
  DescribeSubnetsCommand,
  DescribeSecurityGroupsCommand,
  AuthorizeSecurityGroupIngressCommand,
} from "@aws-sdk/client-ec2";
import {
  IAMClient,
  CreatePolicyCommand,
  CreateRoleCommand,
  CreateInstanceProfileCommand,
  AddRoleToInstanceProfileCommand,
  AttachRolePolicyCommand,
  waitUntilInstanceProfileExists,
} from "@aws-sdk/client-iam";
import { SSMClient, GetParameterCommand } from "@aws-sdk/client-ssm";
import {
  CreateAutoScalingGroupCommand,
  AutoScalingClient,
  AttachLoadBalancerTargetGroupsCommand,
} from "@aws-sdk/client-auto-scaling";
import {
  CreateListenerCommand,
  CreateLoadBalancerCommand,
  CreateTargetGroupCommand,
  ElasticLoadBalancingV2Client,
  waitUntilLoadBalancerAvailable,
} from "@aws-sdk/client-elastic-load-balancing-v2";
```

```
import {
  ScenarioOutput,
  ScenarioInput,
  ScenarioAction,
} from "@aws-doc-sdk-examples/lib/scenario/index.js";
import { saveState } from "@aws-doc-sdk-examples/lib/scenario/steps-common.js";
import { retry } from "@aws-doc-sdk-examples/lib/utils/util-timers.js";

import { MESSAGES, NAMES, RESOURCES_PATH, ROOT } from "./constants.js";
import { initParamsSteps } from "./steps-reset-params.js";

/**
 * @type {import('@aws-doc-sdk-examples/lib/scenario.js').Step[]}
 */
export const deploySteps = [
  new ScenarioOutput("introduction", MESSAGES.introduction, { header: true }),
  new ScenarioInput("confirmDeployment", MESSAGES.confirmDeployment, {
    type: "confirm",
  }),
  new ScenarioAction(
    "handleConfirmDeployment",
    (c) => c.confirmDeployment === false && process.exit(),
  ),
  new ScenarioOutput(
    "creatingTable",
    MESSAGES.creatingTable.replace("${TABLE_NAME}", NAMES.tableName),
  ),
  new ScenarioAction("createTable", async () => {
    const client = new DynamoDBClient({});
    await client.send(
      new CreateTableCommand({
        TableName: NAMES.tableName,
        ProvisionedThroughput: {
          ReadCapacityUnits: 5,
          WriteCapacityUnits: 5,
        },
        AttributeDefinitions: [
          {
            AttributeName: "MediaType",
            AttributeType: "S",
          },
          {
            AttributeName: "ItemId",
```

```

        AttributeType: "N",
      },
    ],
    KeySchema: [
      {
        AttributeName: "MediaType",
        KeyType: "HASH",
      },
      {
        AttributeName: "ItemId",
        KeyType: "RANGE",
      },
    ],
  ]),
);
await waitUntilTableExists({ client }, { TableName: NAMES.tableName });
}),
new ScenarioOutput(
  "createdTable",
  MESSAGES.createdTable.replace("${TABLE_NAME}", NAMES.tableName),
),
new ScenarioOutput(
  "populatingTable",
  MESSAGES.populatingTable.replace("${TABLE_NAME}", NAMES.tableName),
),
new ScenarioAction("populateTable", () => {
  const client = new DynamoDBClient({});
  /**
   * @type {{ default: import("@aws-sdk/client-dynamodb").PutRequest['Item'][] }}
   */
  const recommendations = JSON.parse(
    readFileSync(join(RESOURCES_PATH, "recommendations.json")),
  );

  return client.send(
    new BatchWriteItemCommand({
      RequestItems: {
        [NAMES.tableName]: recommendations.map((item) => ({
          PutRequest: { Item: item },
        })),
      },
    }),
  );
}),
}),
);
}),

```

```
new ScenarioOutput(
  "populatedTable",
  MESSAGES.populatedTable.replace("${TABLE_NAME}", NAMES.tableName),
),
new ScenarioOutput(
  "creatingKeyPair",
  MESSAGES.creatingKeyPair.replace("${KEY_PAIR_NAME}", NAMES.keyPairName),
),
new ScenarioAction("createKeyPair", async () => {
  const client = new EC2Client({});
  const { KeyMaterial } = await client.send(
    new CreateKeyPairCommand({
      KeyName: NAMES.keyPairName,
    }),
  );

  writeFileSync(`${NAMES.keyPairName}.pem`, KeyMaterial, { mode: 0o600 });
}),
new ScenarioOutput(
  "createdKeyPair",
  MESSAGES.createdKeyPair.replace("${KEY_PAIR_NAME}", NAMES.keyPairName),
),
new ScenarioOutput(
  "creatingInstancePolicy",
  MESSAGES.creatingInstancePolicy.replace(
    "${INSTANCE_POLICY_NAME}",
    NAMES.instancePolicyName,
  ),
),
new ScenarioAction("createInstancePolicy", async (state) => {
  const client = new IAMClient({});
  const {
    Policy: { Arn },
  } = await client.send(
    new CreatePolicyCommand({
      PolicyName: NAMES.instancePolicyName,
      PolicyDocument: readFileSync(
        join(RESOURCES_PATH, "instance_policy.json"),
      ),
    }),
  );
  state.instancePolicyArn = Arn;
}),
new ScenarioOutput("createdInstancePolicy", (state) =>
```

```
MESSAGES.createdInstancePolicy
    .replace("${INSTANCE_POLICY_NAME}", NAMES.instancePolicyName)
    .replace("${INSTANCE_POLICY_ARN}", state.instancePolicyArn),
),
new ScenarioOutput(
    "creatingInstanceRole",
    MESSAGES.creatingInstanceRole.replace(
        "${INSTANCE_ROLE_NAME}",
        NAMES.instanceRoleName,
    ),
),
new ScenarioAction("createInstanceRole", () => {
    const client = new IAMClient({});
    return client.send(
        new CreateRoleCommand({
            RoleName: NAMES.instanceRoleName,
            AssumeRolePolicyDocument: readFileSync(
                join(ROOT, "assume-role-policy.json"),
            ),
        }),
    );
}),
new ScenarioOutput(
    "createdInstanceRole",
    MESSAGES.createdInstanceRole.replace(
        "${INSTANCE_ROLE_NAME}",
        NAMES.instanceRoleName,
    ),
),
new ScenarioOutput(
    "attachingPolicyToRole",
    MESSAGES.attachingPolicyToRole
        .replace("${INSTANCE_ROLE_NAME}", NAMES.instanceRoleName)
        .replace("${INSTANCE_POLICY_NAME}", NAMES.instancePolicyName),
),
new ScenarioAction("attachPolicyToRole", async (state) => {
    const client = new IAMClient({});
    await client.send(
        new AttachRolePolicyCommand({
            RoleName: NAMES.instanceRoleName,
            PolicyArn: state.instancePolicyArn,
        }),
    );
}),
}),
```

```
new ScenarioOutput(
  "attachedPolicyToRole",
  MESSAGES.attachedPolicyToRole
    .replace("${INSTANCE_POLICY_NAME}", NAMES.instancePolicyName)
    .replace("${INSTANCE_ROLE_NAME}", NAMES.instanceRoleName),
),
new ScenarioOutput(
  "creatingInstanceProfile",
  MESSAGES.creatingInstanceProfile.replace(
    "${INSTANCE_PROFILE_NAME}",
    NAMES.instanceProfileName,
  ),
),
new ScenarioAction("createInstanceProfile", async (state) => {
  const client = new IAMClient({});
  const {
    InstanceProfile: { Arn },
  } = await client.send(
    new CreateInstanceProfileCommand({
      InstanceProfileName: NAMES.instanceProfileName,
    }),
  );
  state.instanceProfileArn = Arn;

  await waitUntilInstanceProfileExists(
    { client },
    { InstanceProfileName: NAMES.instanceProfileName },
  );
}),
new ScenarioOutput("createdInstanceProfile", (state) =>
  MESSAGES.createdInstanceProfile
    .replace("${INSTANCE_PROFILE_NAME}", NAMES.instanceProfileName)
    .replace("${INSTANCE_PROFILE_ARN}", state.instanceProfileArn),
),
new ScenarioOutput(
  "addingRoleToInstanceProfile",
  MESSAGES.addingRoleToInstanceProfile
    .replace("${INSTANCE_PROFILE_NAME}", NAMES.instanceProfileName)
    .replace("${INSTANCE_ROLE_NAME}", NAMES.instanceRoleName),
),
new ScenarioAction("addRoleToInstanceProfile", () => {
  const client = new IAMClient({});
  return client.send(
    new AddRoleToInstanceProfileCommand({
```



```

        RoleName: NAMES.instanceRoleName,
        InstanceProfileName: NAMES.instanceProfileName,
    })),
    );
  })),
  new ScenarioOutput(
    "addedRoleToInstanceProfile",
    MESSAGES.addedRoleToInstanceProfile
      .replace("${INSTANCE_PROFILE_NAME}", NAMES.instanceProfileName)
      .replace("${INSTANCE_ROLE_NAME}", NAMES.instanceRoleName),
  ),
  ...initParamsSteps,
  new ScenarioOutput("creatingLaunchTemplate", MESSAGES.creatingLaunchTemplate),
  new ScenarioAction("createLaunchTemplate", async () => {
    const ssmClient = new SSMClient({});
    const { Parameter } = await ssmClient.send(
      new GetParameterCommand({
        Name: "/aws/service/ami-amazon-linux-latest/amzn2-ami-hvm-x86_64-gp2",
      })),
    );
    const ec2Client = new EC2Client({});
    await ec2Client.send(
      new CreateLaunchTemplateCommand({
        LaunchTemplateName: NAMES.launchTemplateName,
        LaunchTemplateData: {
          InstanceType: "t3.micro",
          ImageId: Parameter.Value,
          IamInstanceProfile: { Name: NAMES.instanceProfileName },
          UserData: readFileSync(
            join(RESOURCES_PATH, "server_startup_script.sh"),
          ).toString("base64"),
          KeyName: NAMES.keyPairName,
        },
      })),
    );
  })),
  new ScenarioOutput(
    "createdLaunchTemplate",
    MESSAGES.createdLaunchTemplate.replace(
      "${LAUNCH_TEMPLATE_NAME}",
      NAMES.launchTemplateName,
    ),
  ),
  new ScenarioOutput(

```

```

    "creatingAutoScalingGroup",
    MESSAGES.creatingAutoScalingGroup.replace(
      "${AUTO_SCALING_GROUP_NAME}",
      NAMES.autoScalingGroupName,
    ),
  ),
  new ScenarioAction("createAutoScalingGroup", async (state) => {
    const ec2Client = new EC2Client({});
    const { AvailabilityZones } = await ec2Client.send(
      new DescribeAvailabilityZonesCommand({}),
    );
    state.availabilityZoneNames = AvailabilityZones.map((az) => az.ZoneName);
    const autoScalingClient = new AutoScalingClient({});
    await retry({ intervalInMs: 1000, maxRetries: 30 }, () =>
      autoScalingClient.send(
        new CreateAutoScalingGroupCommand({
          AvailabilityZones: state.availabilityZoneNames,
          AutoScalingGroupName: NAMES.autoScalingGroupName,
          LaunchTemplate: {
            LaunchTemplateName: NAMES.launchTemplateName,
            Version: "$Default",
          },
          MinSize: 3,
          MaxSize: 3,
        })),
    );
  })),
  new ScenarioOutput(
    "createdAutoScalingGroup",
    /**
     * @param {{ availabilityZoneNames: string[] }} state
     */
    (state) =>
      MESSAGES.createdAutoScalingGroup
        .replace("${AUTO_SCALING_GROUP_NAME}", NAMES.autoScalingGroupName)
        .replace(
          "${AVAILABILITY_ZONE_NAMES}",
          state.availabilityZoneNames.join(", "),
        ),
  ),
  new ScenarioInput("confirmContinue", MESSAGES.confirmContinue, {
    type: "confirm",
  })),

```

```

new ScenarioOutput("loadBalancer", MESSAGES.loadBalancer),
new ScenarioOutput("gettingVpc", MESSAGES.gettingVpc),
new ScenarioAction("getVpc", async (state) => {
  const client = new EC2Client({});
  const { Vpcs } = await client.send(
    new DescribeVpcsCommand({
      Filters: [{ Name: "is-default", Values: ["true"] }],
    }),
  );
  state.defaultVpc = Vpcs[0].VpcId;
}),
new ScenarioOutput("gotVpc", (state) =>
  MESSAGES.gotVpc.replace("${VPC_ID}", state.defaultVpc),
),
new ScenarioOutput("gettingSubnets", MESSAGES.gettingSubnets),
new ScenarioAction("getSubnets", async (state) => {
  const client = new EC2Client({});
  const { Subnets } = await client.send(
    new DescribeSubnetsCommand({
      Filters: [
        { Name: "vpc-id", Values: [state.defaultVpc] },
        { Name: "availability-zone", Values: state.availabilityZoneNames },
        { Name: "default-for-az", Values: ["true"] },
      ],
    }),
  );
  state.subnets = Subnets.map((subnet) => subnet.SubnetId);
}),
new ScenarioOutput(
  "gotSubnets",
  /**
   * @param {{ subnets: string[] }} state
   */
  (state) =>
    MESSAGES.gotSubnets.replace("${SUBNETS}", state.subnets.join(", ")),
),
new ScenarioOutput(
  "creatingLoadBalancerTargetGroup",
  MESSAGES.creatingLoadBalancerTargetGroup.replace(
    "${TARGET_GROUP_NAME}",
    NAMES.loadBalancerTargetGroupName,
  ),
),
new ScenarioAction("createLoadBalancerTargetGroup", async (state) => {

```

```
const client = new ElasticLoadBalancingV2Client({});
const { TargetGroups } = await client.send(
  new CreateTargetGroupCommand({
    Name: NAMES.loadBalancerTargetGroupName,
    Protocol: "HTTP",
    Port: 80,
    HealthCheckPath: "/healthcheck",
    HealthCheckIntervalSeconds: 10,
    HealthCheckTimeoutSeconds: 5,
    HealthyThresholdCount: 2,
    UnhealthyThresholdCount: 2,
    VpcId: state.defaultVpc,
  }),
);
const targetGroup = TargetGroups[0];
state.targetGroupArn = targetGroup.TargetGroupArn;
state.targetGroupProtocol = targetGroup.Protocol;
state.targetGroupPort = targetGroup.Port;
}),
new ScenarioOutput(
  "createdLoadBalancerTargetGroup",
  MESSAGES.createdLoadBalancerTargetGroup.replace(
    "${TARGET_GROUP_NAME}",
    NAMES.loadBalancerTargetGroupName,
  ),
),
new ScenarioOutput(
  "creatingLoadBalancer",
  MESSAGES.creatingLoadBalancer.replace("${LB_NAME}", NAMES.loadBalancerName),
),
new ScenarioAction("createLoadBalancer", async (state) => {
  const client = new ElasticLoadBalancingV2Client({});
  const { LoadBalancers } = await client.send(
    new CreateLoadBalancerCommand({
      Name: NAMES.loadBalancerName,
      Subnets: state.subnets,
    }),
  );
  state.loadBalancerDns = LoadBalancers[0].DNSName;
  state.loadBalancerArn = LoadBalancers[0].LoadBalancerArn;
  await waitUntilLoadBalancerAvailable(
    { client },
    { Names: [NAMES.loadBalancerName] },
  );
});
```

```
}),
new ScenarioOutput("createdLoadBalancer", (state) =>
  MESSAGES.createdLoadBalancer
    .replace("${LB_NAME}", NAMES.loadBalancerName)
    .replace("${DNS_NAME}", state.loadBalancerDns),
),
new ScenarioOutput(
  "creatingListener",
  MESSAGES.creatingLoadBalancerListener
    .replace("${LB_NAME}", NAMES.loadBalancerName)
    .replace("${TARGET_GROUP_NAME}", NAMES.loadBalancerTargetGroupName),
),
new ScenarioAction("createListener", async (state) => {
  const client = new ElasticLoadBalancingV2Client({});
  const { Listeners } = await client.send(
    new CreateListenerCommand({
      LoadBalancerArn: state.loadBalancerArn,
      Protocol: state.targetGroupProtocol,
      Port: state.targetGroupPort,
      DefaultActions: [
        { Type: "forward", TargetGroupArn: state.targetGroupArn },
      ],
    })
  );
  const listener = Listeners[0];
  state.loadBalancerListenerArn = listener.ListenerArn;
}),
new ScenarioOutput("createdListener", (state) =>
  MESSAGES.createdLoadBalancerListener.replace(
    "${LB_LISTENER_ARN}",
    state.loadBalancerListenerArn,
  ),
),
new ScenarioOutput(
  "attachingLoadBalancerTargetGroup",
  MESSAGES.attachingLoadBalancerTargetGroup
    .replace("${TARGET_GROUP_NAME}", NAMES.loadBalancerTargetGroupName)
    .replace("${AUTO_SCALING_GROUP_NAME}", NAMES.autoScalingGroupName),
),
new ScenarioAction("attachLoadBalancerTargetGroup", async (state) => {
  const client = new AutoScalingClient({});
  await client.send(
    new AttachLoadBalancerTargetGroupsCommand({
      AutoScalingGroupName: NAMES.autoScalingGroupName,
```

```

        TargetGroupARNs: [state.targetGroupArn],
      )),
    );
  )),
  new ScenarioOutput(
    "attachedLoadBalancerTargetGroup",
    MESSAGES.attachedLoadBalancerTargetGroup,
  ),
  new ScenarioOutput("verifyingInboundPort", MESSAGES.verifyingInboundPort),
  new ScenarioAction(
    "verifyInboundPort",
    /**
     *
     * @param {{ defaultSecurityGroup: import('@aws-sdk/client-ec2').SecurityGroup}}
state
    */
    async (state) => {
      const client = new EC2Client({});
      const { SecurityGroups } = await client.send(
        new DescribeSecurityGroupsCommand({
          Filters: [{ Name: "group-name", Values: ["default"] }],
        }),
      );
    };
    if (!SecurityGroups) {
      state.verifyInboundPortError = new Error(MESSAGES.noSecurityGroups);
    }
    state.defaultSecurityGroup = SecurityGroups[0];

    /**
     * @type {string}
     */
    const ipResponse = (await axios.get("http://checkip.amazonaws.com")).data;
    state.myIp = ipResponse.trim();
    const myIpRules = state.defaultSecurityGroup.IpPermissions.filter(
      ({ IpRanges }) =>
        IpRanges.some(
          ({ CidrIp }) =>
            CidrIp.startsWith(state.myIp) || CidrIp === "0.0.0.0/0",
        ),
    )
      .filter(({ IpProtocol }) => IpProtocol === "tcp")
      .filter(({ FromPort }) => FromPort === 80);

    state.myIpRules = myIpRules;
  }

```

```
    },
  ),
  new ScenarioOutput(
    "verifiedInboundPort",
    /**
     * @param {{ myIpRules: any[] }} state
     */
    (state) => {
      if (state.myIpRules.length > 0) {
        return MESSAGES.foundIpRules.replace(
          "${IP_RULES}",
          JSON.stringify(state.myIpRules, null, 2),
        );
      } else {
        return MESSAGES.noIpRules;
      }
    },
  ),
  new ScenarioInput(
    "shouldAddInboundRule",
    /**
     * @param {{ myIpRules: any[] }} state
     */
    (state) => {
      if (state.myIpRules.length > 0) {
        return false;
      } else {
        return MESSAGES.noIpRules;
      }
    },
    { type: "confirm" },
  ),
  new ScenarioAction(
    "addInboundRule",
    /**
     * @param {{ defaultSecurityGroup: import('@aws-sdk/client-ec2').SecurityGroup }} state
     */
    async (state) => {
      if (!state.shouldAddInboundRule) {
        return;
      }

      const client = new EC2Client({});
```

```
    await client.send(
      new AuthorizeSecurityGroupIngressCommand({
        GroupId: state.defaultSecurityGroup.GroupId,
        CidrIp: `${state.myIp}/32`,
        FromPort: 80,
        ToPort: 80,
        IpProtocol: "tcp",
      }),
    );
  },
),
new ScenarioOutput("addedInboundRule", (state) => {
  if (state.shouldAddInboundRule) {
    return MESSAGES.addedInboundRule.replace("${IP_ADDRESS}", state.myIp);
  } else {
    return false;
  }
}),
new ScenarioOutput("verifyingEndpoint", (state) =>
  MESSAGES.verifyingEndpoint.replace("${DNS_NAME}", state.loadBalancerDns),
),
new ScenarioAction("verifyEndpoint", async (state) => {
  try {
    const response = await retry({ intervalInMs: 2000, maxRetries: 30 }, () =>
      axios.get(`http://${state.loadBalancerDns}`),
    );
    state.endpointResponse = JSON.stringify(response.data, null, 2);
  } catch (e) {
    state.verifyEndpointError = e;
  }
}),
new ScenarioOutput("verifiedEndpoint", (state) => {
  if (state.verifyEndpointError) {
    console.error(state.verifyEndpointError);
  } else {
    return MESSAGES.verifiedEndpoint.replace(
      "${ENDPOINT_RESPONSE}",
      state.endpointResponse,
    );
  }
}),
saveState,
];
```


Create steps to run the demo.

```
import { readFileSync } from "node:fs";
import { join } from "node:path";

import axios from "axios";

import {
  DescribeTargetGroupsCommand,
  DescribeTargetHealthCommand,
  ElasticLoadBalancingV2Client,
} from "@aws-sdk/client-elastic-load-balancing-v2";
import {
  DescribeInstanceInformationCommand,
  PutParameterCommand,
  SSMClient,
  SendCommandCommand,
} from "@aws-sdk/client-ssm";
import {
  IAMClient,
  CreatePolicyCommand,
  CreateRoleCommand,
  AttachRolePolicyCommand,
  CreateInstanceProfileCommand,
  AddRoleToInstanceProfileCommand,
  waitUntilInstanceProfileExists,
} from "@aws-sdk/client-iam";
import {
  AutoScalingClient,
  DescribeAutoScalingGroupsCommand,
  TerminateInstanceInAutoScalingGroupCommand,
} from "@aws-sdk/client-auto-scaling";
import {
  DescribeIamInstanceProfileAssociationsCommand,
  EC2Client,
  RebootInstancesCommand,
  ReplaceIamInstanceProfileAssociationCommand,
} from "@aws-sdk/client-ec2";

import {
  ScenarioAction,
```

```
ScenarioInput,
ScenarioOutput,
} from "@aws-doc-sdk-examples/lib/scenario/scenario.js";
import { retry } from "@aws-doc-sdk-examples/lib/utils/util-timers.js";

import { MESSAGES, NAMES, RESOURCES_PATH } from "./constants.js";
import { findLoadBalancer } from "./shared.js";

const getRecommendation = new ScenarioAction(
  "getRecommendation",
  async (state) => {
    const loadBalancer = await findLoadBalancer(NAMES.loadBalancerName);
    if (loadBalancer) {
      state.loadBalancerDnsName = loadBalancer.DNSName;
      try {
        state.recommendation = (
          await axios.get(`http://${state.loadBalancerDnsName}`)
        ).data;
      } catch (e) {
        state.recommendation = e instanceof Error ? e.message : e;
      }
    } else {
      throw new Error(MESSAGES.demoFindLoadBalancerError);
    }
  },
);

const getRecommendationResult = new ScenarioOutput(
  "getRecommendationResult",
  (state) =>
    `Recommendation:\n${JSON.stringify(state.recommendation, null, 2)}`,
  { preformatted: true },
);

const getHealthCheck = new ScenarioAction("getHealthCheck", async (state) => {
  const client = new ElasticLoadBalancingV2Client({});
  const { TargetGroups } = await client.send(
    new DescribeTargetGroupsCommand({
      Names: [NAMES.loadBalancerTargetGroupName],
    }),
  );
});

const { TargetHealthDescriptions } = await client.send(
  new DescribeTargetHealthCommand({
```

```
        TargetGroupArn: TargetGroups[0].TargetGroupArn,
    })),
  );
  state.targetHealthDescriptions = TargetHealthDescriptions;
});

const getHealthCheckResult = new ScenarioOutput(
  "getHealthCheckResult",
  /**
   * @param {{ targetHealthDescriptions: import('@aws-sdk/client-elastic-load-
  balancing-v2').TargetHealthDescription[]}} state
   */
  (state) => {
    const status = state.targetHealthDescriptions
      .map((th) => `${th.Target.Id}: ${th.TargetHealth.State}`)
      .join("\n");
    return `Health check:\n${status}`;
  },
  { preformatted: true },
);

const loadBalancerLoop = new ScenarioAction(
  "loadBalancerLoop",
  getRecommendation.action,
  {
    whileConfig: {
      whileFn: ({ loadBalancerCheck }) => loadBalancerCheck,
      input: new ScenarioInput(
        "loadBalancerCheck",
        MESSAGES.demoLoadBalancerCheck,
        {
          type: "confirm",
        },
      ),
      output: getRecommendationResult,
    },
  },
);

const healthCheckLoop = new ScenarioAction(
  "healthCheckLoop",
  getHealthCheck.action,
  {
    whileConfig: {
```

```
    whileFn: ({ healthCheck }) => healthCheck,
    input: new ScenarioInput("healthCheck", MESSAGES.demoHealthCheck, {
      type: "confirm",
    }),
    output: getHealthCheckResult,
  },
},
);

const statusSteps = [
  getRecommendation,
  getRecommendationResult,
  getHealthCheck,
  getHealthCheckResult,
];

/**
 * @type {import('@aws-doc-sdk-examples/lib/scenario.js').Step[]}
 */
export const demoSteps = [
  new ScenarioOutput("header", MESSAGES.demoHeader, { header: true }),
  new ScenarioOutput("sanityCheck", MESSAGES.demoSanityCheck),
  ...statusSteps,
  new ScenarioInput(
    "brokenDependencyConfirmation",
    MESSAGES.demoBrokenDependencyConfirmation,
    { type: "confirm" },
  ),
  new ScenarioAction("brokenDependency", async (state) => {
    if (!state.brokenDependencyConfirmation) {
      process.exit();
    } else {
      const client = new SSMClient({});
      state.badTableName = `fake-table-${Date.now()}`;
      await client.send(
        new PutParameterCommand({
          Name: NAMES.ssmTableNameKey,
          Value: state.badTableName,
          Overwrite: true,
          Type: "String",
        }),
      );
    }
  })
],
);
```

```
new ScenarioOutput("testBrokenDependency", (state) =>
  MESSAGES.demoTestBrokenDependency.replace(
    "${TABLE_NAME}",
    state.badTableName,
  ),
),
...statusSteps,
new ScenarioInput(
  "staticResponseConfirmation",
  MESSAGES.demoStaticResponseConfirmation,
  { type: "confirm" },
),
new ScenarioAction("staticResponse", async (state) => {
  if (!state.staticResponseConfirmation) {
    process.exit();
  } else {
    const client = new SSMClient({});
    await client.send(
      new PutParameterCommand({
        Name: NAMES.ssmFailureResponseKey,
        Value: "static",
        Overwrite: true,
        Type: "String",
      }),
    );
  }
}),
new ScenarioOutput("testStaticResponse", MESSAGES.demoTestStaticResponse),
...statusSteps,
new ScenarioInput(
  "badCredentialsConfirmation",
  MESSAGES.demoBadCredentialsConfirmation,
  { type: "confirm" },
),
new ScenarioAction("badCredentialsExit", (state) => {
  if (!state.badCredentialsConfirmation) {
    process.exit();
  }
}),
new ScenarioAction("fixDynamoDBName", async () => {
  const client = new SSMClient({});
  await client.send(
    new PutParameterCommand({
      Name: NAMES.ssmTableNameKey,
```

```

        Value: NAMES.tableName,
        Overwrite: true,
        Type: "String",
    )),
    );
  })),
  new ScenarioAction(
    "badCredentials",
    /**
     * @param {{ targetInstance: import('@aws-sdk/client-auto-scaling').Instance }}
state
     */
    async (state) => {
      await createSsmOnlyInstanceProfile();
      const autoScalingClient = new AutoScalingClient({});
      const { AutoScalingGroups } = await autoScalingClient.send(
        new DescribeAutoScalingGroupsCommand({
          AutoScalingGroupNames: [NAMES.autoScalingGroupName],
        })),
      );
      state.targetInstance = AutoScalingGroups[0].Instances[0];
      const ec2Client = new EC2Client({});
      const { IamInstanceProfileAssociations } = await ec2Client.send(
        new DescribeIamInstanceProfileAssociationsCommand({
          Filters: [
            { Name: "instance-id", Values: [state.targetInstance.InstanceId] },
          ],
        })),
      );
      state.instanceProfileAssociationId =
        IamInstanceProfileAssociations[0].AssociationId;
      await retry({ intervalInMs: 1000, maxRetries: 30 }, () =>
        ec2Client.send(
          new ReplaceIamInstanceProfileAssociationCommand({
            AssociationId: state.instanceProfileAssociationId,
            IamInstanceProfile: { Name: NAMES.ssmOnlyInstanceProfileName },
          })),
        ),
      );

      await ec2Client.send(
        new RebootInstancesCommand({
          InstanceIds: [state.targetInstance.InstanceId],
        })),
      );
    }
  )
);

```

```
);

const ssmClient = new SSMClient({});
await retry({ intervalInMs: 20000, maxRetries: 15 }, async () => {
  const { InstanceInformationList } = await ssmClient.send(
    new DescribeInstanceInformationCommand({}),
  );

  const instance = InstanceInformationList.find(
    (info) => info.InstanceId === state.targetInstance.InstanceId,
  );

  if (!instance) {
    throw new Error("Instance not found.");
  }
});

await ssmClient.send(
  new SendCommandCommand({
    InstanceIds: [state.targetInstance.InstanceId],
    DocumentName: "AWS-RunShellScript",
    Parameters: { commands: ["cd / && sudo python3 server.py 80"] },
  }),
);
},
),
new ScenarioOutput(
  "testBadCredentials",
  /**
   * @param {{ targetInstance: import('@aws-sdk/client-ssm').InstanceInformation}}
state
   */
  (state) =>
    MESSAGES.demoTestBadCredentials.replace(
      "${INSTANCE_ID}",
      state.targetInstance.InstanceId,
    ),
),
loadBalancerLoop,
new ScenarioInput(
  "deepHealthCheckConfirmation",
  MESSAGES.demoDeepHealthCheckConfirmation,
  { type: "confirm" },
),
```

```

new ScenarioAction("deepHealthCheckExit", (state) => {
  if (!state.deepHealthCheckConfirmation) {
    process.exit();
  }
}),
new ScenarioAction("deepHealthCheck", async () => {
  const client = new SSMClient({});
  await client.send(
    new PutParameterCommand({
      Name: NAMES.ssmHealthCheckKey,
      Value: "deep",
      Overwrite: true,
      Type: "String",
    }),
  );
}),
new ScenarioOutput("testDeepHealthCheck", MESSAGES.demoTestDeepHealthCheck),
healthCheckLoop,
loadBalancerLoop,
new ScenarioInput(
  "killInstanceConfirmation",
  /**
   * @param {{ targetInstance: import('@aws-sdk/client-
  ssm').InstanceInformation }} state
   */
  (state) =>
    MESSAGES.demoKillInstanceConfirmation.replace(
      "${INSTANCE_ID}",
      state.targetInstance.InstanceId,
    ),
  { type: "confirm" },
),
new ScenarioAction("killInstanceExit", (state) => {
  if (!state.killInstanceConfirmation) {
    process.exit();
  }
}),
new ScenarioAction(
  "killInstance",
  /**
   * @param {{ targetInstance: import('@aws-sdk/client-
  ssm').InstanceInformation }} state
   */
  async (state) => {

```



```
const client = new AutoScalingClient({});
await client.send(
  new TerminateInstanceInAutoScalingGroupCommand({
    InstanceId: state.targetInstance.InstanceId,
    ShouldDecrementDesiredCapacity: false,
  }),
);
},
),
new ScenarioOutput("testKillInstance", MESSAGES.demoTestKillInstance),
healthCheckLoop,
loadBalancerLoop,
new ScenarioInput("failOpenConfirmation", MESSAGES.demoFailOpenConfirmation, {
  type: "confirm",
}),
new ScenarioAction("failOpenExit", (state) => {
  if (!state.failOpenConfirmation) {
    process.exit();
  }
}),
new ScenarioAction("failOpen", () => {
  const client = new SSMClient({});
  return client.send(
    new PutParameterCommand({
      Name: NAMES.ssmTableNameKey,
      Value: `fake-table-${Date.now()}`,
      Overwrite: true,
      Type: "String",
    }),
  );
}),
new ScenarioOutput("testFailOpen", MESSAGES.demoFailOpenTest),
healthCheckLoop,
loadBalancerLoop,
new ScenarioInput(
  "resetTableConfirmation",
  MESSAGES.demoResetTableConfirmation,
  { type: "confirm" },
),
new ScenarioAction("resetTableExit", (state) => {
  if (!state.resetTableConfirmation) {
    process.exit();
  }
}),
},
```

```
new ScenarioAction("resetTable", async () => {
  const client = new SSMClient({});
  await client.send(
    new PutParameterCommand({
      Name: NAMES.ssmTableNameKey,
      Value: NAMES.tableName,
      Overwrite: true,
      Type: "String",
    }),
  );
}),
new ScenarioOutput("testResetTable", MESSAGES.demoTestResetTable),
healthCheckLoop,
loadBalancerLoop,
];

async function createSsmOnlyInstanceProfile() {
  const iamClient = new IAMClient({});
  const { Policy } = await iamClient.send(
    new CreatePolicyCommand({
      PolicyName: NAMES.ssmOnlyPolicyName,
      PolicyDocument: readFileSync(
        join(RESOURCES_PATH, "ssm_only_policy.json"),
      ),
    }),
  );
  await iamClient.send(
    new CreateRoleCommand({
      RoleName: NAMES.ssmOnlyRoleName,
      AssumeRolePolicyDocument: JSON.stringify({
        Version: "2012-10-17",
        Statement: [
          {
            Effect: "Allow",
            Principal: { Service: "ec2.amazonaws.com" },
            Action: "sts:AssumeRole",
          },
        ],
      }),
    ),
  );
  await iamClient.send(
    new AttachRolePolicyCommand({
      RoleName: NAMES.ssmOnlyRoleName,
```

```
        PolicyArn: Policy.Arn,
    })),
  );
  await iamClient.send(
    new AttachRolePolicyCommand({
      RoleName: NAMES.ssmOnlyRoleName,
      PolicyArn: "arn:aws:iam::aws:policy/AmazonSSMManagedInstanceCore",
    })),
  );
  const { InstanceProfile } = await iamClient.send(
    new CreateInstanceProfileCommand({
      InstanceProfileName: NAMES.ssmOnlyInstanceProfileName,
    })),
  );
  await waitUntilInstanceProfileExists(
    { client: iamClient },
    { InstanceProfileName: NAMES.ssmOnlyInstanceProfileName },
  );
  await iamClient.send(
    new AddRoleToInstanceProfileCommand({
      InstanceProfileName: NAMES.ssmOnlyInstanceProfileName,
      RoleName: NAMES.ssmOnlyRoleName,
    })),
  );

  return InstanceProfile;
}
```

Create steps to destroy all of the resources.

```
import { unlinkSync } from "node:fs";

import { DynamoDBClient, DeleteTableCommand } from "@aws-sdk/client-dynamodb";
import {
  EC2Client,
  DeleteKeyPairCommand,
  DeleteLaunchTemplateCommand,
  RevokeSecurityGroupIngressCommand,
} from "@aws-sdk/client-ec2";
import {
  IAMClient,
  DeleteInstanceProfileCommand,
```

```
    RemoveRoleFromInstanceProfileCommand,
    DeletePolicyCommand,
    DeleteRoleCommand,
    DetachRolePolicyCommand,
    paginateListPolicies,
  } from "@aws-sdk/client-iam";
import {
  AutoScalingClient,
  DeleteAutoScalingGroupCommand,
  TerminateInstanceInAutoScalingGroupCommand,
  UpdateAutoScalingGroupCommand,
  paginateDescribeAutoScalingGroups,
} from "@aws-sdk/client-auto-scaling";
import {
  DeleteLoadBalancerCommand,
  DeleteTargetGroupCommand,
  DescribeTargetGroupsCommand,
  ElasticLoadBalancingV2Client,
} from "@aws-sdk/client-elastic-load-balancing-v2";

import {
  ScenarioOutput,
  ScenarioInput,
  ScenarioAction,
} from "@aws-doc-sdk-examples/lib/scenario/index.js";
import { loadState } from "@aws-doc-sdk-examples/lib/scenario/steps-common.js";
import { retry } from "@aws-doc-sdk-examples/lib/utils/util-timers.js";

import { MESSAGES, NAMES } from "./constants.js";
import { findLoadBalancer } from "./shared.js";

/**
 * @type {import('@aws-doc-sdk-examples/lib/scenario.js').Step[]}
 */
export const destroySteps = [
  loadState,
  new ScenarioInput("destroy", MESSAGES.destroy, { type: "confirm" }),
  new ScenarioAction(
    "abort",
    (state) => state.destroy === false && process.exit(),
  ),
  new ScenarioAction("deleteTable", async (c) => {
    try {
      const client = new DynamoDBClient({});
```

```
    await client.send(new DeleteTableCommand({ TableName: NAMES.tableName }));
  } catch (e) {
    c.deleteTableError = e;
  }
}),
new ScenarioOutput("deleteTableResult", (state) => {
  if (state.deleteTableError) {
    console.error(state.deleteTableError);
    return MESSAGES.deleteTableError.replace(
      "${TABLE_NAME}",
      NAMES.tableName,
    );
  } else {
    return MESSAGES.deletedTable.replace("${TABLE_NAME}", NAMES.tableName);
  }
}),
new ScenarioAction("deleteKeyPair", async (state) => {
  try {
    const client = new EC2Client({});
    await client.send(
      new DeleteKeyPairCommand({ KeyName: NAMES.keyPairName }),
    );
    unlinkSync(`${NAMES.keyPairName}.pem`);
  } catch (e) {
    state.deleteKeyPairError = e;
  }
}),
new ScenarioOutput("deleteKeyPairResult", (state) => {
  if (state.deleteKeyPairError) {
    console.error(state.deleteKeyPairError);
    return MESSAGES.deleteKeyPairError.replace(
      "${KEY_PAIR_NAME}",
      NAMES.keyPairName,
    );
  } else {
    return MESSAGES.deletedKeyPair.replace(
      "${KEY_PAIR_NAME}",
      NAMES.keyPairName,
    );
  }
}),
new ScenarioAction("detachPolicyFromRole", async (state) => {
  try {
    const client = new IAMClient({});
```

```
const policy = await findPolicy(NAMES.instancePolicyName);

if (!policy) {
  state.detachPolicyFromRoleError = new Error(
    `Policy ${NAMES.instancePolicyName} not found.`
  );
} else {
  await client.send(
    new DetachRolePolicyCommand({
      RoleName: NAMES.instanceRoleName,
      PolicyArn: policy.Arn,
    })
  );
}
} catch (e) {
  state.detachPolicyFromRoleError = e;
}
}),
new ScenarioOutput("detachedPolicyFromRole", (state) => {
  if (state.detachPolicyFromRoleError) {
    console.error(state.detachPolicyFromRoleError);
    return MESSAGES.detachPolicyFromRoleError
      .replace("${INSTANCE_POLICY_NAME}", NAMES.instancePolicyName)
      .replace("${INSTANCE_ROLE_NAME}", NAMES.instanceRoleName);
  } else {
    return MESSAGES.detachedPolicyFromRole
      .replace("${INSTANCE_POLICY_NAME}", NAMES.instancePolicyName)
      .replace("${INSTANCE_ROLE_NAME}", NAMES.instanceRoleName);
  }
}),
new ScenarioAction("deleteInstancePolicy", async (state) => {
  const client = new IAMClient({});
  const policy = await findPolicy(NAMES.instancePolicyName);

  if (!policy) {
    state.deletePolicyError = new Error(
      `Policy ${NAMES.instancePolicyName} not found.`
    );
  } else {
    return client.send(
      new DeletePolicyCommand({
        PolicyArn: policy.Arn,
      })
    );
  }
});
```

```
    }
  })),
  new ScenarioOutput("deletePolicyResult", (state) => {
    if (state.deletePolicyError) {
      console.error(state.deletePolicyError);
      return MESSAGES.deletePolicyError.replace(
        "${INSTANCE_POLICY_NAME}",
        NAMES.instancePolicyName,
      );
    } else {
      return MESSAGES.deletedPolicy.replace(
        "${INSTANCE_POLICY_NAME}",
        NAMES.instancePolicyName,
      );
    }
  })),
  new ScenarioAction("removeRoleFromInstanceProfile", async (state) => {
    try {
      const client = new IAMClient({});
      await client.send(
        new RemoveRoleFromInstanceProfileCommand({
          RoleName: NAMES.instanceRoleName,
          InstanceProfileName: NAMES.instanceProfileName,
        }),
      );
    } catch (e) {
      state.removeRoleFromInstanceProfileError = e;
    }
  })),
  new ScenarioOutput("removeRoleFromInstanceProfileResult", (state) => {
    if (state.removeRoleFromInstanceProfile) {
      console.error(state.removeRoleFromInstanceProfileError);
      return MESSAGES.removeRoleFromInstanceProfileError
        .replace("${INSTANCE_PROFILE_NAME}", NAMES.instanceProfileName)
        .replace("${INSTANCE_ROLE_NAME}", NAMES.instanceRoleName);
    } else {
      return MESSAGES.removedRoleFromInstanceProfile
        .replace("${INSTANCE_PROFILE_NAME}", NAMES.instanceProfileName)
        .replace("${INSTANCE_ROLE_NAME}", NAMES.instanceRoleName);
    }
  })),
  new ScenarioAction("deleteInstanceRole", async (state) => {
    try {
      const client = new IAMClient({});
```

```
    await client.send(
      new DeleteRoleCommand({
        RoleName: NAMES.instanceRoleName,
      }),
    );
  } catch (e) {
    state.deleteInstanceRoleError = e;
  }
}),
new ScenarioOutput("deleteInstanceRoleResult", (state) => {
  if (state.deleteInstanceRoleError) {
    console.error(state.deleteInstanceRoleError);
    return MESSAGES.deleteInstanceRoleError.replace(
      "${INSTANCE_ROLE_NAME}",
      NAMES.instanceRoleName,
    );
  } else {
    return MESSAGES.deletedInstanceRole.replace(
      "${INSTANCE_ROLE_NAME}",
      NAMES.instanceRoleName,
    );
  }
}),
new ScenarioAction("deleteInstanceProfile", async (state) => {
  try {
    const client = new IAMClient({});
    await client.send(
      new DeleteInstanceProfileCommand({
        InstanceProfileName: NAMES.instanceProfileName,
      }),
    );
  } catch (e) {
    state.deleteInstanceProfileError = e;
  }
}),
new ScenarioOutput("deleteInstanceProfileResult", (state) => {
  if (state.deleteInstanceProfileError) {
    console.error(state.deleteInstanceProfileError);
    return MESSAGES.deleteInstanceProfileError.replace(
      "${INSTANCE_PROFILE_NAME}",
      NAMES.instanceProfileName,
    );
  } else {
    return MESSAGES.deletedInstanceProfile.replace(
```



```
        "${INSTANCE_PROFILE_NAME}",
        NAMES.instanceProfileName,
    );
}
}),
new ScenarioAction("deleteAutoScalingGroup", async (state) => {
    try {
        await terminateGroupInstances(NAMES.autoScalingGroupName);
        await retry({ intervalInMs: 60000, maxRetries: 60 }, async () => {
            await deleteAutoScalingGroup(NAMES.autoScalingGroupName);
        });
    } catch (e) {
        state.deleteAutoScalingGroupError = e;
    }
}),
new ScenarioOutput("deleteAutoScalingGroupResult", (state) => {
    if (state.deleteAutoScalingGroupError) {
        console.error(state.deleteAutoScalingGroupError);
        return MESSAGES.deleteAutoScalingGroupError.replace(
            "${AUTO_SCALING_GROUP_NAME}",
            NAMES.autoScalingGroupName,
        );
    } else {
        return MESSAGES.deletedAutoScalingGroup.replace(
            "${AUTO_SCALING_GROUP_NAME}",
            NAMES.autoScalingGroupName,
        );
    }
}),
new ScenarioAction("deleteLaunchTemplate", async (state) => {
    const client = new EC2Client({});
    try {
        await client.send(
            new DeleteLaunchTemplateCommand({
                LaunchTemplateName: NAMES.launchTemplateName,
            }),
        );
    } catch (e) {
        state.deleteLaunchTemplateError = e;
    }
}),
new ScenarioOutput("deleteLaunchTemplateResult", (state) => {
    if (state.deleteLaunchTemplateError) {
        console.error(state.deleteLaunchTemplateError);
    }
});
```

```
    return MESSAGES.deleteLaunchTemplateError.replace(
      "${LAUNCH_TEMPLATE_NAME}",
      NAMES.launchTemplateName,
    );
  } else {
    return MESSAGES.deletedLaunchTemplate.replace(
      "${LAUNCH_TEMPLATE_NAME}",
      NAMES.launchTemplateName,
    );
  }
}),
new ScenarioAction("deleteLoadBalancer", async (state) => {
  try {
    const client = new ElasticLoadBalancingV2Client({});
    const loadBalancer = await findLoadBalancer(NAMES.loadBalancerName);
    await client.send(
      new DeleteLoadBalancerCommand({
        LoadBalancerArn: loadBalancer.LoadBalancerArn,
      }),
    );
    await retry({ intervalInMs: 1000, maxRetries: 60 }, async () => {
      const lb = await findLoadBalancer(NAMES.loadBalancerName);
      if (lb) {
        throw new Error("Load balancer still exists.");
      }
    });
  } catch (e) {
    state.deleteLoadBalancerError = e;
  }
}),
new ScenarioOutput("deleteLoadBalancerResult", (state) => {
  if (state.deleteLoadBalancerError) {
    console.error(state.deleteLoadBalancerError);
    return MESSAGES.deleteLoadBalancerError.replace(
      "${LB_NAME}",
      NAMES.loadBalancerName,
    );
  } else {
    return MESSAGES.deletedLoadBalancer.replace(
      "${LB_NAME}",
      NAMES.loadBalancerName,
    );
  }
}),

```

```
new ScenarioAction("deleteLoadBalancerTargetGroup", async (state) => {
  const client = new ElasticLoadBalancingV2Client({});
  try {
    const { TargetGroups } = await client.send(
      new DescribeTargetGroupsCommand({
        Names: [NAMES.loadBalancerTargetGroupName],
      }),
    );

    await retry({ intervalInMs: 1000, maxRetries: 30 }, () =>
      client.send(
        new DeleteTargetGroupCommand({
          TargetGroupArn: TargetGroups[0].TargetGroupArn,
        }),
      ),
    );
  } catch (e) {
    state.deleteLoadBalancerTargetGroupError = e;
  }
}),
new ScenarioOutput("deleteLoadBalancerTargetGroupResult", (state) => {
  if (state.deleteLoadBalancerTargetGroupError) {
    console.error(state.deleteLoadBalancerTargetGroupError);
    return MESSAGES.deleteLoadBalancerTargetGroupError.replace(
      "${TARGET_GROUP_NAME}",
      NAMES.loadBalancerTargetGroupName,
    );
  } else {
    return MESSAGES.deletedLoadBalancerTargetGroup.replace(
      "${TARGET_GROUP_NAME}",
      NAMES.loadBalancerTargetGroupName,
    );
  }
}),
new ScenarioAction("detachSsmOnlyRoleFromProfile", async (state) => {
  try {
    const client = new IAMClient({});
    await client.send(
      new RemoveRoleFromInstanceProfileCommand({
        InstanceProfileName: NAMES.ssmOnlyInstanceProfileName,
        RoleName: NAMES.ssmOnlyRoleName,
      }),
    );
  } catch (e) {
```

```
    state.detachSsmOnlyRoleFromProfileError = e;
  }
}),
new ScenarioOutput("detachSsmOnlyRoleFromProfileResult", (state) => {
  if (state.detachSsmOnlyRoleFromProfileError) {
    console.error(state.detachSsmOnlyRoleFromProfileError);
    return MESSAGES.detachSsmOnlyRoleFromProfileError
      .replace("${ROLE_NAME}", NAMES.ssmOnlyRoleName)
      .replace("${PROFILE_NAME}", NAMES.ssmOnlyInstanceProfileName);
  } else {
    return MESSAGES.detachedSsmOnlyRoleFromProfile
      .replace("${ROLE_NAME}", NAMES.ssmOnlyRoleName)
      .replace("${PROFILE_NAME}", NAMES.ssmOnlyInstanceProfileName);
  }
}),
new ScenarioAction("detachSsmOnlyCustomRolePolicy", async (state) => {
  try {
    const iamClient = new IAMClient({});
    const ssmOnlyPolicy = await findPolicy(NAMES.ssmOnlyPolicyName);
    await iamClient.send(
      new DetachRolePolicyCommand({
        RoleName: NAMES.ssmOnlyRoleName,
        PolicyArn: ssmOnlyPolicy.Arn,
      })),
    );
  } catch (e) {
    state.detachSsmOnlyCustomRolePolicyError = e;
  }
}),
new ScenarioOutput("detachSsmOnlyCustomRolePolicyResult", (state) => {
  if (state.detachSsmOnlyCustomRolePolicyError) {
    console.error(state.detachSsmOnlyCustomRolePolicyError);
    return MESSAGES.detachSsmOnlyCustomRolePolicyError
      .replace("${ROLE_NAME}", NAMES.ssmOnlyRoleName)
      .replace("${POLICY_NAME}", NAMES.ssmOnlyPolicyName);
  } else {
    return MESSAGES.detachedSsmOnlyCustomRolePolicy
      .replace("${ROLE_NAME}", NAMES.ssmOnlyRoleName)
      .replace("${POLICY_NAME}", NAMES.ssmOnlyPolicyName);
  }
}),
new ScenarioAction("detachSsmOnlyAWSRolePolicy", async (state) => {
  try {
    const iamClient = new IAMClient({});
```

```
    await iamClient.send(
      new DetachRolePolicyCommand({
        RoleName: NAMES.ssmOnlyRoleName,
        PolicyArn: "arn:aws:iam::aws:policy/AmazonSSMManagedInstanceCore",
      }),
    );
  } catch (e) {
    state.detachSsmOnlyAWSRolePolicyError = e;
  }
}),
new ScenarioOutput("detachSsmOnlyAWSRolePolicyResult", (state) => {
  if (state.detachSsmOnlyAWSRolePolicyError) {
    console.error(state.detachSsmOnlyAWSRolePolicyError);
    return MESSAGES.detachSsmOnlyAWSRolePolicyError
      .replace("${ROLE_NAME}", NAMES.ssmOnlyRoleName)
      .replace("${POLICY_NAME}", "AmazonSSMManagedInstanceCore");
  } else {
    return MESSAGES.detachedSsmOnlyAWSRolePolicy
      .replace("${ROLE_NAME}", NAMES.ssmOnlyRoleName)
      .replace("${POLICY_NAME}", "AmazonSSMManagedInstanceCore");
  }
}),
new ScenarioAction("deleteSsmOnlyInstanceProfile", async (state) => {
  try {
    const iamClient = new IAMClient({});
    await iamClient.send(
      new DeleteInstanceProfileCommand({
        InstanceProfileName: NAMES.ssmOnlyInstanceProfileName,
      }),
    );
  } catch (e) {
    state.deleteSsmOnlyInstanceProfileError = e;
  }
}),
new ScenarioOutput("deleteSsmOnlyInstanceProfileResult", (state) => {
  if (state.deleteSsmOnlyInstanceProfileError) {
    console.error(state.deleteSsmOnlyInstanceProfileError);
    return MESSAGES.deleteSsmOnlyInstanceProfileError.replace(
      "${INSTANCE_PROFILE_NAME}",
      NAMES.ssmOnlyInstanceProfileName,
    );
  } else {
    return MESSAGES.deletedSsmOnlyInstanceProfile.replace(
      "${INSTANCE_PROFILE_NAME}",

```

```
        NAMES.ssmOnlyInstanceProfileName,
    );
}
}),
new ScenarioAction("deleteSsmOnlyPolicy", async (state) => {
    try {
        const iamClient = new IAMClient({});
        const ssmOnlyPolicy = await findPolicy(NAMES.ssmOnlyPolicyName);
        await iamClient.send(
            new DeletePolicyCommand({
                PolicyArn: ssmOnlyPolicy.Arn,
            }),
        );
    } catch (e) {
        state.deleteSsmOnlyPolicyError = e;
    }
}),
new ScenarioOutput("deleteSsmOnlyPolicyResult", (state) => {
    if (state.deleteSsmOnlyPolicyError) {
        console.error(state.deleteSsmOnlyPolicyError);
        return MESSAGES.deleteSsmOnlyPolicyError.replace(
            "${POLICY_NAME}",
            NAMES.ssmOnlyPolicyName,
        );
    } else {
        return MESSAGES.deletedSsmOnlyPolicy.replace(
            "${POLICY_NAME}",
            NAMES.ssmOnlyPolicyName,
        );
    }
}),
new ScenarioAction("deleteSsmOnlyRole", async (state) => {
    try {
        const iamClient = new IAMClient({});
        await iamClient.send(
            new DeleteRoleCommand({
                RoleName: NAMES.ssmOnlyRoleName,
            }),
        );
    } catch (e) {
        state.deleteSsmOnlyRoleError = e;
    }
}),
new ScenarioOutput("deleteSsmOnlyRoleResult", (state) => {
```

```

    if (state.deleteSsmOnlyRoleError) {
      console.error(state.deleteSsmOnlyRoleError);
      return MESSAGES.deleteSsmOnlyRoleError.replace(
        "${ROLE_NAME}",
        NAMES.ssmOnlyRoleName,
      );
    } else {
      return MESSAGES.deletedSsmOnlyRole.replace(
        "${ROLE_NAME}",
        NAMES.ssmOnlyRoleName,
      );
    }
  })),
  new ScenarioAction(
    "revokeSecurityGroupIngress",
    async (
      /** @type {{ myIp: string, defaultSecurityGroup: { GroupId: string } }} */
      state,
    ) => {
      const ec2Client = new EC2Client({});

      try {
        await ec2Client.send(
          new RevokeSecurityGroupIngressCommand({
            GroupId: state.defaultSecurityGroup.GroupId,
            CidrIp: `${state.myIp}/32`,
            FromPort: 80,
            ToPort: 80,
            IpProtocol: "tcp",
          }),
        );
      } catch (e) {
        state.revokeSecurityGroupIngressError = e;
      }
    },
  ),
  new ScenarioOutput("revokeSecurityGroupIngressResult", (state) => {
    if (state.revokeSecurityGroupIngressError) {
      console.error(state.revokeSecurityGroupIngressError);
      return MESSAGES.revokeSecurityGroupIngressError.replace(
        "${IP}",
        state.myIp,
      );
    } else {

```

```
        return MESSAGES.revokedSecurityGroupIngress.replace("${IP}", state.myIp);
    }
  })),
];

/**
 * @param {string} policyName
 */
async function findPolicy(policyName) {
  const client = new IAMClient({});
  const paginatedPolicies = paginateListPolicies({ client }, {});
  for await (const page of paginatedPolicies) {
    const policy = page.Policies.find((p) => p.PolicyName === policyName);
    if (policy) {
      return policy;
    }
  }
}

/**
 * @param {string} groupName
 */
async function deleteAutoScalingGroup(groupName) {
  const client = new AutoScalingClient({});
  try {
    await client.send(
      new DeleteAutoScalingGroupCommand({
        AutoScalingGroupName: groupName,
      }),
    );
  } catch (err) {
    if (!(err instanceof Error)) {
      throw err;
    } else {
      console.log(err.name);
      throw err;
    }
  }
}

/**
 * @param {string} groupName
 */
async function terminateGroupInstances(groupName) {
```



```
const autoScalingClient = new AutoScalingClient({});
const group = await findAutoScalingGroup(groupName);
await autoScalingClient.send(
  new UpdateAutoScalingGroupCommand({
    AutoScalingGroupName: group.AutoScalingGroupName,
    MinSize: 0,
  }),
);
for (const i of group.Instances) {
  await retry({ intervalInMs: 1000, maxRetries: 30 }, () =>
    autoScalingClient.send(
      new TerminateInstanceInAutoScalingGroupCommand({
        InstanceId: i.InstanceId,
        ShouldDecrementDesiredCapacity: true,
      }),
    ),
);
}
}

async function findAutoScalingGroup(groupName) {
  const client = new AutoScalingClient({});
  const paginatedGroups = paginateDescribeAutoScalingGroups({ client }, {});
  for await (const page of paginatedGroups) {
    const group = page.AutoScalingGroups.find(
      (g) => g.AutoScalingGroupName === groupName,
    );
    if (group) {
      return group;
    }
  }
  throw new Error(`Auto scaling group ${groupName} not found.`);
}
```

- For API details, see the following topics in *AWS SDK for JavaScript API Reference*.
 - [AttachLoadBalancerTargetGroups](#)
 - [CreateAutoScalingGroup](#)
 - [CreateInstanceProfile](#)
 - [CreateLaunchTemplate](#)
 - [CreateListener](#)

- [CreateLoadBalancer](#)
- [CreateTargetGroup](#)
- [DeleteAutoScalingGroup](#)
- [DeleteInstanceProfile](#)
- [DeleteLaunchTemplate](#)
- [DeleteLoadBalancer](#)
- [DeleteTargetGroup](#)
- [DescribeAutoScalingGroups](#)
- [DescribeAvailabilityZones](#)
- [DescribeIamInstanceProfileAssociations](#)
- [DescribeInstances](#)
- [DescribeLoadBalancers](#)
- [DescribeSubnets](#)
- [DescribeTargetGroups](#)
- [DescribeTargetHealth](#)
- [DescribeVpcs](#)
- [RebootInstances](#)
- [ReplacelamInstanceProfileAssociation](#)
- [TerminateInstanceInAutoScalingGroup](#)
- [UpdateAutoScalingGroup](#)

EventBridge examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with EventBridge.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Scenarios are code examples that show you how to accomplish specific tasks by calling multiple functions within a service or combined with other AWS services.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Topics

- [Actions](#)
- [Scenarios](#)

Actions

PutEvents

The following code example shows how to use PutEvents.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Import the SDK and client modules and call the API.

```
import {
  EventBridgeClient,
  PutEventsCommand,
} from "@aws-sdk/client-eventbridge";

export const putEvents = async (
  source = "eventbridge.integration.test",
  detailType = "greeting",
  resources = [],
) => {
  const client = new EventBridgeClient({});

  const response = await client.send(
    new PutEventsCommand({
      Entries: [
        {
          Detail: JSON.stringify({ greeting: "Hello there." }),
          DetailType: detailType,
          Resources: resources,
          Source: source,
```

```
    },
  ],
}),
);

console.log("PutEvents response:");
console.log(response);
// PutEvents response:
// {
//   '$metadata': {
//     httpStatusCode: 200,
//     requestId: '3d0df73d-dcea-4a23-ae0d-f5556a3ac109',
//     extendedRequestId: undefined,
//     cfId: undefined,
//     attempts: 1,
//     totalRetryDelay: 0
//   },
//   Entries: [ { EventId: '51620841-5af4-6402-d9bc-b77734991eb5' } ],
//   FailedEntryCount: 0
// }

return response;
};
```

- For API details, see [PutEvents](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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 ebevents = new AWS.EventBridge({ apiVersion: "2015-10-07" });
```

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

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

- For API details, see [PutEvents](#) in *AWS SDK for JavaScript API Reference*.

PutRule

The following code example shows how to use PutRule.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Import the SDK and client modules and call the API.

```
import { EventBridgeClient, PutRuleCommand } from "@aws-sdk/client-eventbridge";

export const putRule = async (
  ruleName = "some-rule",
  source = "some-source",
```

```
) => {
  const client = new EventBridgeClient({});

  const response = await client.send(
    new PutRuleCommand({
      Name: ruleName,
      EventPattern: JSON.stringify({ source: [source] }),
      State: "ENABLED",
      EventBusName: "default",
    }),
  );

  console.log("PutRule response:");
  console.log(response);
  // PutRule response:
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: 'd7292ced-1544-421b-842f-596326bc7072',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  //   RuleArn: 'arn:aws:events:us-east-1:xxxxxxxxxxxx:rule/
EventBridgeTestRule-1696280037720'
  // }
  return response;
};
```

- For API details, see [PutRule](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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 ebevents = new AWS.EventBridge({ apiVersion: "2015-10-07" });

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

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

- For API details, see [PutRule](#) in *AWS SDK for JavaScript API Reference*.

PutTargets

The following code example shows how to use PutTargets.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Import the SDK and client modules and call the API.

```
import {
  EventBridgeClient,
  PutTargetsCommand,
```

```
} from "@aws-sdk/client-eventbridge";

export const putTarget = async (
  existingRuleName = "some-rule",
  targetArn = "arn:aws:lambda:us-east-1:000000000000:function:test-func",
  uniqueId = Date.now().toString(),
) => {
  const client = new EventBridgeClient({});
  const response = await client.send(
    new PutTargetsCommand({
      Rule: existingRuleName,
      Targets: [
        {
          Arn: targetArn,
          Id: uniqueId,
        },
      ],
    }),
  );

  console.log("PutTargets response:");
  console.log(response);
  // PutTargets response:
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: 'f5b23b9a-2c17-45c1-ad5c-f926c3692e3d',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  //   FailedEntries: [],
  //   FailedEntryCount: 0
  // }

  return response;
};
```

- For API details, see [PutTargets](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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 ebevents = new AWS.EventBridge({ apiVersion: "2015-10-07" });

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

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

- For API details, see [PutTargets](#) in *AWS SDK for JavaScript API Reference*.

Scenarios

Use scheduled events to invoke a Lambda function

The following code example shows how to create an AWS Lambda function invoked by an Amazon EventBridge scheduled event.

SDK for JavaScript (v3)

Shows how to create an Amazon EventBridge scheduled event that invokes an AWS Lambda function. Configure EventBridge to use a cron expression to schedule when the Lambda function is invoked. In this example, you create a Lambda function by using the Lambda JavaScript runtime API. This example invokes different AWS services to perform a specific use case. This example demonstrates how to create an app that sends a mobile text message to your employees that congratulates them at the one year anniversary date.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

This example is also available in the [AWS SDK for JavaScript v3 developer guide](#).

Services used in this example

- DynamoDB
- EventBridge
- Lambda
- Amazon SNS

AWS Glue examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with AWS Glue.

Basics are code examples that show you how to perform the essential operations within a service.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Get started

Hello AWS Glue

The following code examples show how to get started using AWS Glue.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { ListJobsCommand, GlueClient } from "@aws-sdk/client-glue";

const client = new GlueClient({});

export const main = async () => {
  const command = new ListJobsCommand({});

  const { JobNames } = await client.send(command);
  const formattedJobNames = JobNames.join("\n");
  console.log("Job names: ");
  console.log(formattedJobNames);
  return JobNames;
};
```

- For API details, see [ListJobs](#) in *AWS SDK for JavaScript API Reference*.

Topics

- [Basics](#)
- [Actions](#)

Basics

Learn the basics

The following code example shows how to:

- Create a crawler that crawls a public Amazon S3 bucket and generates a database of CSV-formatted metadata.
- List information about databases and tables in your AWS Glue Data Catalog.
- Create a job to extract CSV data from the S3 bucket, transform the data, and load JSON-formatted output into another S3 bucket.
- List information about job runs, view transformed data, and clean up resources.

For more information, see [Tutorial: Getting started with AWS Glue Studio](#).

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create and run a crawler that crawls a public Amazon Simple Storage Service (Amazon S3) bucket and generates a metadata database that describes the CSV-formatted data it finds.

```
const createCrawler = (name, role, dbName, tablePrefix, s3TargetPath) => {
  const client = new GlueClient({});

  const command = new CreateCrawlerCommand({
    Name: name,
    Role: role,
    DatabaseName: dbName,
    TablePrefix: tablePrefix,
    Targets: {
      S3Targets: [{ Path: s3TargetPath }],
    },
  });

  return client.send(command);
};

const getCrawler = (name) => {
  const client = new GlueClient({});

  const command = new GetCrawlerCommand({
```

```
    Name: name,
  });

  return client.send(command);
};

const startCrawler = (name) => {
  const client = new GlueClient({});

  const command = new StartCrawlerCommand({
    Name: name,
  });

  return client.send(command);
};

const crawlerExists = async ({ getCrawler }, crawlerName) => {
  try {
    await getCrawler(crawlerName);
    return true;
  } catch {
    return false;
  }
};

/**
 * @param {{ createCrawler: import('../../../../../actions/create-crawler.js').createCrawler}} actions
 */
const makeCreateCrawlerStep = (actions) => async (context) => {
  if (await crawlerExists(actions, process.env.CRAWLER_NAME)) {
    log("Crawler already exists. Skipping creation.");
  } else {
    await actions.createCrawler(
      process.env.CRAWLER_NAME,
      process.env.ROLE_NAME,
      process.env.DATABASE_NAME,
      process.env.TABLE_PREFIX,
      process.env.S3_TARGET_PATH,
    );

    log("Crawler created successfully.", { type: "success" });
  }
}
```

```
    return { ...context };
  };

  /**
   * @param {(name: string) => Promise<import('@aws-sdk/client-glue').GetCrawlerCommandOutput>} getCrawler
   * @param {string} crawlerName
   */
  const waitForCrawler = async (getCrawler, crawlerName) => {
    const waitTimeInSeconds = 30;
    const { Crawler } = await getCrawler(crawlerName);

    if (!Crawler) {
      throw new Error(`Crawler with name ${crawlerName} not found.`);
    }

    if (Crawler.State === "READY") {
      return;
    }

    log(`Crawler is ${Crawler.State}. Waiting ${waitTimeInSeconds} seconds...`);
    await wait(waitTimeInSeconds);
    return waitForCrawler(getCrawler, crawlerName);
  };

  const makeStartCrawlerStep =
    ({ startCrawler, getCrawler }) =>
    async (context) => {
      log("Starting crawler.");
      await startCrawler(process.env.CRAWLER_NAME);
      log("Crawler started.", { type: "success" });

      log("Waiting for crawler to finish running. This can take a while.");
      await waitForCrawler(getCrawler, process.env.CRAWLER_NAME);
      log("Crawler ready.", { type: "success" });

      return { ...context };
    };
  };
};
```

List information about databases and tables in your AWS Glue Data Catalog.

```
const getDatabase = (name) => {
```

```
const client = new GlueClient({});

const command = new GetDatabaseCommand({
  Name: name,
});

return client.send(command);
};

const getTables = (databaseName) => {
  const client = new GlueClient({});

  const command = new GetTablesCommand({
    DatabaseName: databaseName,
  });

  return client.send(command);
};

const makeGetDatabaseStep =
  ({ getDatabase }) =>
  async (context) => {
    const {
      Database: { Name },
    } = await getDatabase(process.env.DATABASE_NAME);
    log(`Database: ${Name}`);
    return { ...context };
  };

/**
 * @param {{ getTables: () => Promise<import('@aws-sdk/client-
 * glue').GetTablesCommandOutput>}} config
 */
const makeGetTablesStep =
  ({ getTables }) =>
  async (context) => {
    const { TableList } = await getTables(process.env.DATABASE_NAME);
    log("Tables:");
    log(TableList.map((table) => `  • ${table.Name}\n`));
    return { ...context };
  };
};
```

Create and run a job that extracts CSV data from the source Amazon S3 bucket, transforms it by removing and renaming fields, and loads JSON-formatted output into another Amazon S3 bucket.

```
const createJob = (name, role, scriptBucketName, scriptKey) => {
  const client = new GlueClient({});

  const command = new CreateJobCommand({
    Name: name,
    Role: role,
    Command: {
      Name: "glueetl",
      PythonVersion: "3",
      ScriptLocation: `s3://${scriptBucketName}/${scriptKey}`,
    },
    GlueVersion: "3.0",
  });

  return client.send(command);
};

const startJobRun = (jobName, dbName, tableName, bucketName) => {
  const client = new GlueClient({});

  const command = new StartJobRunCommand({
    JobName: jobName,
    Arguments: {
      "--input_database": dbName,
      "--input_table": tableName,
      "--output_bucket_url": `s3://${bucketName}/`,
    },
  });

  return client.send(command);
};

const makeCreateJobStep =
  ({ createJob }) =>
  async (context) => {
    log("Creating Job.");
    await createJob(
      process.env.JOB_NAME,
      process.env.ROLE_NAME,
```



```
        process.env.BUCKET_NAME,
        process.env.PYTHON_SCRIPT_KEY,
    );
    log("Job created.", { type: "success" });

    return { ...context };
};

/**
 * @param {(name: string, runId: string) => Promise<import('@aws-sdk/client-glue').GetJobRunCommandOutput> } getJobRun
 * @param {string} jobName
 * @param {string} jobRunId
 */
const waitForJobRun = async (getJobRun, jobName, jobRunId) => {
    const waitTimeInSeconds = 30;
    const { JobRun } = await getJobRun(jobName, jobRunId);

    if (!JobRun) {
        throw new Error(`Job run with id ${jobRunId} not found.`);
    }

    switch (JobRun.JobRunState) {
        case "FAILED":
        case "TIMEOUT":
        case "STOPPED":
            throw new Error(
                `Job ${JobRun.JobRunState}. Error: ${JobRun.ErrorMessage}`,
            );
        case "RUNNING":
            break;
        case "SUCCEEDED":
            return;
        default:
            throw new Error(`Unknown job run state: ${JobRun.JobRunState}`);
    }

    log(
        `Job ${JobRun.JobRunState}. Waiting ${waitTimeInSeconds} more seconds...`,
    );
    await wait(waitTimeInSeconds);
    return waitForJobRun(getJobRun, jobName, jobRunId);
};
```

```
/**
 * @param {{ prompter: { prompt: () => Promise<{ shouldOpen: boolean }>} }} context
 */
const promptToOpen = async (context) => {
  const { shouldOpen } = await context.prompter.prompt({
    name: "shouldOpen",
    type: "confirm",
    message: "Open the output bucket in your browser?",
  });

  if (shouldOpen) {
    return open(
      `https://s3.console.aws.amazon.com/s3/buckets/${process.env.BUCKET_NAME} to
view the output.` ,
    );
  }
};

const makeStartJobRunStep =
({ startJobRun, getJobRun }) =>
async (context) => {
  log("Starting job.");
  const { JobRunId } = await startJobRun(
    process.env.JOB_NAME,
    process.env.DATABASE_NAME,
    process.env.TABLE_NAME,
    process.env.BUCKET_NAME,
  );
  log("Job started.", { type: "success" });

  log("Waiting for job to finish running. This can take a while.");
  await waitForJobRun(getJobRun, process.env.JOB_NAME, JobRunId);
  log("Job run succeeded.", { type: "success" });

  await promptToOpen(context);

  return { ...context };
};
```

List information about job runs and view some of the transformed data.

```
const getJobRuns = (jobName) => {
```

```
const client = new GlueClient({});
const command = new GetJobRunsCommand({
  JobName: jobName,
});

return client.send(command);
};

const getJobRun = (jobName, jobRunId) => {
  const client = new GlueClient({});
  const command = new GetJobRunCommand({
    JobName: jobName,
    RunId: jobRunId,
  });

  return client.send(command);
};

/**
 * @typedef {{ prompter: { prompt: () => Promise<{jobName: string}> } }} Context
 */

/**
 * @typedef {() => Promise<import('@aws-sdk/client-glue').GetJobRunCommandOutput>}
getJobRun
 */

/**
 * @typedef {() => Promise<import('@aws-sdk/client-glue').GetJobRunsCommandOutput>}
getJobRuns
 */

/**
 *
 * @param {getJobRun} getJobRun
 * @param {string} jobName
 * @param {string} jobRunId
 */
const logJobRunDetails = async (getJobRun, jobName, jobRunId) => {
  const { JobRun } = await getJobRun(jobName, jobRunId);
  log(JobRun, { type: "object" });
};

/**
```

```
*
* @param {{getJobRuns: getJobRuns, getJobRun: getJobRun }} funcs
*/
const makePickJobRunStep =
  ({ getJobRuns, getJobRun }) =>
  async (** @type { Context } */ context) => {
    if (context.selectedJobName) {
      const { JobRuns } = await getJobRuns(context.selectedJobName);

      const { jobRunId } = await context.prompter.prompt({
        name: "jobRunId",
        type: "list",
        message: "Select a job run to see details.",
        choices: JobRuns.map((run) => run.Id),
      });

      logJobRunDetails(getJobRun, context.selectedJobName, jobRunId);
    }

    return { ...context };
  };
```

Delete all resources created by the demo.

```
const deleteJob = (jobName) => {
  const client = new GlueClient({});

  const command = new DeleteJobCommand({
    JobName: jobName,
  });

  return client.send(command);
};

const deleteTable = (databaseName, tableName) => {
  const client = new GlueClient({});

  const command = new DeleteTableCommand({
    DatabaseName: databaseName,
    Name: tableName,
  });
};
```

```
    return client.send(command);
  };

const deleteDatabase = (databaseName) => {
  const client = new GlueClient({});

  const command = new DeleteDatabaseCommand({
    Name: databaseName,
  });

  return client.send(command);
};

const deleteCrawler = (crawlerName) => {
  const client = new GlueClient({});

  const command = new DeleteCrawlerCommand({
    Name: crawlerName,
  });

  return client.send(command);
};

/**
 *
 * @param {import('.././../actions/delete-job.js').deleteJob} deleteJobFn
 * @param {string[]} jobNames
 * @param {{ prompter: { prompt: () => Promise<any> }}} context
 */
const handleDeleteJobs = async (deleteJobFn, jobNames, context) => {
  /**
   * @type {{ selectedJobNames: string[] }}
   */
  const { selectedJobNames } = await context.prompter.prompt({
    name: "selectedJobNames",
    type: "checkbox",
    message: "Let's clean up jobs. Select jobs to delete.",
    choices: jobNames,
  });

  if (selectedJobNames.length === 0) {
    log("No jobs selected.");
  } else {
    log("Deleting jobs.");
  }
};
```

```
    await Promise.all(
      selectedJobNames.map((n) => deleteJobFn(n).catch(console.error)),
    );
    log("Jobs deleted.", { type: "success" });
  }
};

/**
 * @param {{
 *   listJobs: import('.././../actions/list-jobs.js').listJobs,
 *   deleteJob: import('.././../actions/delete-job.js').deleteJob
 * }} config
 */
const makeCleanUpJobsStep =
  ({ listJobs, deleteJob }) =>
  async (context) => {
    const { JobNames } = await listJobs();
    if (JobNames.length > 0) {
      await handleDeleteJobs(deleteJob, JobNames, context);
    }

    return { ...context };
  };

/**
 * @param {import('.././../actions/delete-table.js').deleteTable} deleteTable
 * @param {string} databaseName
 * @param {string[]} tableNames
 */
const deleteTables = (deleteTable, databaseName, tableNames) =>
  Promise.all(
    tableNames.map((tableName) =>
      deleteTable(databaseName, tableName).catch(console.error),
    ),
  );

/**
 * @param {{
 *   getTables: import('.././../actions/get-tables.js').getTables,
 *   deleteTable: import('.././../actions/delete-table.js').deleteTable
 * }} config
 */
const makeCleanUpTablesStep =
  ({ getTables, deleteTable }) =>
```

```
/**
 * @param {{ prompter: { prompt: () => Promise<any>}}} context
 */
async (context) => {
  const { TableList } = await getTables(process.env.DATABASE_NAME).catch(
    () => ({ TableList: null }),
  );

  if (TableList && TableList.length > 0) {
    /**
     * @type {{ tableNames: string[] }}
     */
    const { tableNames } = await context.prompter.prompt({
      name: "tableNames",
      type: "checkbox",
      message: "Let's clean up tables. Select tables to delete.",
      choices: TableList.map((t) => t.Name),
    });

    if (tableNames.length === 0) {
      log("No tables selected.");
    } else {
      log("Deleting tables.");
      await deleteTables(deleteTable, process.env.DATABASE_NAME, tableNames);
      log("Tables deleted.", { type: "success" });
    }
  }

  return { ...context };
};

/**
 * @param {import('.././../actions/delete-database.js').deleteDatabase}
deleteDatabase
 * @param {string[]} databaseNames
 */
const deleteDatabases = (deleteDatabase, databaseNames) =>
  Promise.all(
    databaseNames.map((dbName) => deleteDatabase(dbName).catch(console.error)),
  );

/**
 * @param {{
 *   getDatabases: import('.././../actions/get-databases.js').getDatabases
```

```
* deleteDatabase: import('.././.././actions/delete-database.js').deleteDatabase
* }} config
*/
const makeCleanUpDatabasesStep =
  ({ getDatabases, deleteDatabase }) =>
  /**
   * @param {{ prompter: { prompt: () => Promise<any>}} context
   */
  async (context) => {
    const { DatabaseList } = await getDatabases();

    if (DatabaseList.length > 0) {
      /** @type {{ dbNames: string[] }} */
      const { dbNames } = await context.prompter.prompt({
        name: "dbNames",
        type: "checkbox",
        message: "Let's clean up databases. Select databases to delete.",
        choices: DatabaseList.map((db) => db.Name),
      });

      if (dbNames.length === 0) {
        log("No databases selected.");
      } else {
        log("Deleting databases.");
        await deleteDatabases(deleteDatabase, dbNames);
        log("Databases deleted.", { type: "success" });
      }
    }

    return { ...context };
  };

const cleanUpCrawlerStep = async (context) => {
  log(`Deleting crawler.`);

  try {
    await deleteCrawler(process.env.CRAWLER_NAME);
    log("Crawler deleted.", { type: "success" });
  } catch (err) {
    if (err.name === "EntityNotFoundException") {
      log(`Crawler is already deleted.`);
    } else {
      throw err;
    }
  }
}
```



```
    }  
  
    return { ...context };  
};
```

- For API details, see the following topics in *AWS SDK for JavaScript API Reference*.
 - [CreateCrawler](#)
 - [CreateJob](#)
 - [DeleteCrawler](#)
 - [DeleteDatabase](#)
 - [DeleteJob](#)
 - [DeleteTable](#)
 - [GetCrawler](#)
 - [GetDatabase](#)
 - [GetDatabases](#)
 - [GetJob](#)
 - [GetJobRun](#)
 - [GetJobRuns](#)
 - [GetTables](#)
 - [ListJobs](#)
 - [StartCrawler](#)
 - [StartJobRun](#)

Actions

CreateCrawler

The following code example shows how to use `CreateCrawler`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const createCrawler = (name, role, dbName, tablePrefix, s3TargetPath) => {
  const client = new GlueClient({});

  const command = new CreateCrawlerCommand({
    Name: name,
    Role: role,
    DatabaseName: dbName,
    TablePrefix: tablePrefix,
    Targets: {
      S3Targets: [{ Path: s3TargetPath }],
    },
  });

  return client.send(command);
};
```

- For API details, see [CreateCrawler](#) in *AWS SDK for JavaScript API Reference*.

CreateJob

The following code example shows how to use CreateJob.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const createJob = (name, role, scriptBucketName, scriptKey) => {
```

```
const client = new GlueClient({});

const command = new CreateJobCommand({
  Name: name,
  Role: role,
  Command: {
    Name: "glueetl",
    PythonVersion: "3",
    ScriptLocation: `s3://${scriptBucketName}/${scriptKey}`,
  },
  GlueVersion: "3.0",
});

return client.send(command);
};
```

- For API details, see [CreateJob](#) in *AWS SDK for JavaScript API Reference*.

DeleteCrawler

The following code example shows how to use DeleteCrawler.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const deleteCrawler = (crawlerName) => {
  const client = new GlueClient({});

  const command = new DeleteCrawlerCommand({
    Name: crawlerName,
  });

  return client.send(command);
};
```

- For API details, see [DeleteCrawler](#) in *AWS SDK for JavaScript API Reference*.

DeleteDatabase

The following code example shows how to use DeleteDatabase.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const deleteDatabase = (databaseName) => {
  const client = new GlueClient({});

  const command = new DeleteDatabaseCommand({
    Name: databaseName,
  });

  return client.send(command);
};
```

- For API details, see [DeleteDatabase](#) in *AWS SDK for JavaScript API Reference*.

DeleteJob

The following code example shows how to use DeleteJob.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const deleteJob = (jobName) => {
```

```
const client = new GlueClient({});

const command = new DeleteJobCommand({
  JobName: jobName,
});

return client.send(command);
};
```

- For API details, see [DeleteJob](#) in *AWS SDK for JavaScript API Reference*.

DeleteTable

The following code example shows how to use DeleteTable.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const deleteTable = (databaseName, tableName) => {
  const client = new GlueClient({});

  const command = new DeleteTableCommand({
    DatabaseName: databaseName,
    Name: tableName,
  });

  return client.send(command);
};
```

- For API details, see [DeleteTable](#) in *AWS SDK for JavaScript API Reference*.

GetCrawler

The following code example shows how to use GetCrawler.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const getCrawler = (name) => {
  const client = new GlueClient({});

  const command = new GetCrawlerCommand({
    Name: name,
  });

  return client.send(command);
};
```

- For API details, see [GetCrawler](#) in *AWS SDK for JavaScript API Reference*.

GetDatabase

The following code example shows how to use GetDatabase.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const getDatabase = (name) => {
  const client = new GlueClient({});

  const command = new GetDatabaseCommand({
    Name: name,
  });
};
```

```
    return client.send(command);  
  };
```

- For API details, see [GetDatabase](#) in *AWS SDK for JavaScript API Reference*.

GetDatabases

The following code example shows how to use GetDatabases.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const getDatabases = () => {  
  const client = new GlueClient({});  
  
  const command = new GetDatabasesCommand({});  
  
  return client.send(command);  
};
```

- For API details, see [GetDatabases](#) in *AWS SDK for JavaScript API Reference*.

GetJob

The following code example shows how to use GetJob.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const getJob = (jobName) => {
  const client = new GlueClient({});

  const command = new GetJobCommand({
    JobName: jobName,
  });

  return client.send(command);
};
```

- For API details, see [GetJob](#) in *AWS SDK for JavaScript API Reference*.

GetJobRun

The following code example shows how to use GetJobRun.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const getJobRun = (jobName, jobRunId) => {
  const client = new GlueClient({});
  const command = new GetJobRunCommand({
    JobName: jobName,
    RunId: jobRunId,
  });

  return client.send(command);
};
```

- For API details, see [GetJobRun](#) in *AWS SDK for JavaScript API Reference*.

GetJobRuns

The following code example shows how to use GetJobRuns.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const getJobRuns = (jobName) => {
  const client = new GlueClient({});
  const command = new GetJobRunsCommand({
    JobName: jobName,
  });

  return client.send(command);
};
```

- For API details, see [GetJobRuns](#) in *AWS SDK for JavaScript API Reference*.

GetTables

The following code example shows how to use GetTables.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const getTables = (databaseName) => {
  const client = new GlueClient({});
```

```
const command = new GetTablesCommand({
  DatabaseName: databaseName,
});

return client.send(command);
};
```

- For API details, see [GetTables](#) in *AWS SDK for JavaScript API Reference*.

ListJobs

The following code example shows how to use `ListJobs`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const listJobs = () => {
  const client = new GlueClient({});

  const command = new ListJobsCommand({});

  return client.send(command);
};
```

- For API details, see [ListJobs](#) in *AWS SDK for JavaScript API Reference*.

StartCrawler

The following code example shows how to use `StartCrawler`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const startCrawler = (name) => {
  const client = new GlueClient({});

  const command = new StartCrawlerCommand({
    Name: name,
  });

  return client.send(command);
};
```

- For API details, see [StartCrawler](#) in *AWS SDK for JavaScript API Reference*.

StartJobRun

The following code example shows how to use StartJobRun.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const startJobRun = (jobName, dbName, tableName, bucketName) => {
  const client = new GlueClient({});

  const command = new StartJobRunCommand({
    JobName: jobName,
    Arguments: {
      "--input_database": dbName,
    },
  });

  return client.send(command);
};
```

```
    "--input_table": tableName,
    "--output_bucket_url": `s3://${bucketName}/`,
  },
});

return client.send(command);
};
```

- For API details, see [StartJobRun](#) in *AWS SDK for JavaScript API Reference*.

HealthImaging examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with HealthImaging.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Scenarios are code examples that show you how to accomplish specific tasks by calling multiple functions within a service or combined with other AWS services.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Get started

Hello HealthImaging

The following code examples show how to get started using HealthImaging.

SDK for JavaScript (v3)

```
import {
  ListDatastoresCommand,
  MedicalImagingClient,
} from "@aws-sdk/client-medical-imaging";

// When no region or credentials are provided, the SDK will use the
// region and credentials from the local AWS config.
const client = new MedicalImagingClient({});
```

```
export const helloMedicalImaging = async () => {
  const command = new ListDatastoresCommand({});

  const { datastoreSummaries } = await client.send(command);
  console.log("Datastores: ");
  console.log(datastoreSummaries.map((item) => item.datastoreName).join("\n"));
  return datastoreSummaries;
};
```

- For API details, see [ListDatastores](#) in *AWS SDK for JavaScript API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Topics

- [Actions](#)
- [Scenarios](#)

Actions

CopyImageSet

The following code example shows how to use CopyImageSet.

SDK for JavaScript (v3)

Utility function to copy an image set.

```
import {CopyImageSetCommand} from "@aws-sdk/client-medical-imaging";
import {medicalImagingClient} from "../libs/medicalImagingClient.js";

/**
 * @param {string} datastoreId - The ID of the data store.
 * @param {string} imageSetId - The source image set ID.
 * @param {string} sourceVersionId - The source version ID.
 * @param {string} destinationImageSetId - The optional ID of the destination image
 set.
```

```
* @param {string} destinationVersionId - The optional version ID of the destination
image set.
* @param {boolean} force - Force the copy action.
* @param {[string]} copySubsets - A subset of instance IDs to copy.
*/
export const copyImageSet = async (
  datastoreId = "xxxxxxxxxxxx",
  imageSetId = "xxxxxxxxxxxx",
  sourceVersionId = "1",
  destinationImageSetId = "",
  destinationVersionId = "",
  force = false,
  copySubsets = []
) => {
  try {
    const params = {
      datastoreId: datastoreId,
      sourceImageSetId: imageSetId,
      copyImageSetInformation: {
        sourceImageSet: {latestVersionId: sourceVersionId},
      },
      force: force
    };
    if (destinationImageSetId !== "" && destinationVersionId !== "") {
      params.copyImageSetInformation.destinationImageSet = {
        imageSetId: destinationImageSetId,
        latestVersionId: destinationVersionId,
      };
    }

    if (copySubsets.length > 0) {
      let copySubsetsJson;
      copySubsetsJson = {
        SchemaVersion: 1.1,
        Study: {
          Series: {
            imageSetId: {
              Instances: {}
            }
          }
        }
      };
    }

    for (let i = 0; i < copySubsets.length; i++) {
```

```

        copySubsetsJson.Study.Series.imageSetId.Instances[
            copySubsets[i]
        ] = {};
    }

    params.copyImageSetInformation.dicomCopies = copySubsetsJson;
}

const response = await medicalImagingClient.send(
    new CopyImageSetCommand(params)
);
console.log(response);
// {
//   '$metadata': {
//     httpStatusCode: 200,
//     requestId: 'd9b219ce-cc48-4a44-a5b2-c5c3068f1ee8',
//     extendedRequestId: undefined,
//     cfId: undefined,
//     attempts: 1,
//     totalRetryDelay: 0
//   },
//   datastoreId: 'xxxxxxxxxxxxxxxx',
//   destinationImageSetProperties: {
//     createdAt: 2023-09-27T19:46:21.824Z,
//     imageSetArn: 'arn:aws:medical-imaging:us-
east-1:xxxxxxxxxxxx:datastore/xxxxxxxxxxxxxxxx/imageset/xxxxxxxxxxxxxxxx',
//     imageSetId: 'xxxxxxxxxxxxxxxx',
//     imageSetState: 'LOCKED',
//     imageSetWorkflowStatus: 'COPYING',
//     latestVersionId: '1',
//     updatedAt: 2023-09-27T19:46:21.824Z
//   },
//   sourceImageSetProperties: {
//     createdAt: 2023-09-22T14:49:26.427Z,
//     imageSetArn: 'arn:aws:medical-imaging:us-
east-1:xxxxxxxxxxxx:datastore/xxxxxxxxxxxxxxxx/imageset/xxxxxxxxxxxxxxxx',
//     imageSetId: 'xxxxxxxxxxxxxxxx',
//     imageSetState: 'LOCKED',
//     imageSetWorkflowStatus: 'COPYING_WITH_READ_ONLY_ACCESS',
//     latestVersionId: '4',
//     updatedAt: 2023-09-27T19:46:21.824Z
//   }
// }
return response;

```

```
    } catch (err) {  
        console.error(err);  
    }  
};
```

Copy an image set without a destination.

```
await copyImageSet(  
    "12345678901234567890123456789012",  
    "12345678901234567890123456789012",  
    "1"  
);
```

Copy an image set with a destination.

```
await copyImageSet(  
    "12345678901234567890123456789012",  
    "12345678901234567890123456789012",  
    "1",  
    "12345678901234567890123456789012",  
    "1",  
    false,  
);
```

Copy a subset of an image set with a destination and force the copy.

```
await copyImageSet(  
    "12345678901234567890123456789012",  
    "12345678901234567890123456789012",  
    "1",  
    "12345678901234567890123456789012",  
    "1",  
    true,  
    ["12345678901234567890123456789012", "11223344556677889900112233445566"]  
);
```


- For API details, see [CopyImageSet](#) in *AWS SDK for JavaScript API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

CreateDatastore

The following code example shows how to use CreateDatastore.

SDK for JavaScript (v3)

```
import { CreateDatastoreCommand } from "@aws-sdk/client-medical-imaging";
import { medicalImagingClient } from "../libs/medicalImagingClient.js";

/**
 * @param {string} datastoreName - The name of the data store to create.
 */
export const createDatastore = async (datastoreName = "DATASTORE_NAME") => {
  const response = await medicalImagingClient.send(
    new CreateDatastoreCommand({ datastoreName: datastoreName })
  );
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: 'a71cd65f-2382-49bf-b682-f9209d8d399b',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  //   datastoreId: 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',
  //   datastoreStatus: 'CREATING'
  // }
  return response;
};
```

- For API details, see [CreateDatastore](#) in *AWS SDK for JavaScript API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

DeleteDatastore

The following code example shows how to use DeleteDatastore.

SDK for JavaScript (v3)

```
import { DeleteDatastoreCommand } from "@aws-sdk/client-medical-imaging";
import { medicalImagingClient } from "../libs/medicalImagingClient.js";

/**
 * @param {string} datastoreId - The ID of the data store to delete.
 */
export const deleteDatastore = async (datastoreId = "DATASTORE_ID") => {
  const response = await medicalImagingClient.send(
    new DeleteDatastoreCommand({ datastoreId })
  );
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: 'f5beb409-678d-48c9-9173-9a001ee1ebb1',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  //   datastoreId: 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',
  //   datastoreStatus: 'DELETING'
  // }

  return response;
};
```

- For API details, see [DeleteDatastore](#) in *AWS SDK for JavaScript API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

DeleteImageSet

The following code example shows how to use DeleteImageSet.

SDK for JavaScript (v3)

```
import { DeleteImageSetCommand } from "@aws-sdk/client-medical-imaging";
import { medicalImagingClient } from "../libs/medicalImagingClient.js";

/**
 * @param {string} datastoreId - The data store ID.
 * @param {string} imageSetId - The image set ID.
 */
export const deleteImageSet = async (
  datastoreId = "xxxxxxxxxxxxxxxxxxxx",
  imageSetId = "xxxxxxxxxxxxxxxxxxxx"
) => {
  const response = await medicalImagingClient.send(
    new DeleteImageSetCommand({
      datastoreId: datastoreId,
      imageSetId: imageSetId,
    })
  );
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: '6267bbd2-eaa5-4a50-8ee8-8fddf535cf73',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  // }
```

```

//    datastoreId: 'xxxxxxxxxxxxxxxxxxxx',
//    imageSetId: 'xxxxxxxxxxxxxxxxxxxx',
//    imageSetState: 'LOCKED',
//    imageSetWorkflowStatus: 'DELETING'
// }
return response;
};

```

- For API details, see [DeleteImageSet](#) in *AWS SDK for JavaScript API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

GetDICOMImportJob

The following code example shows how to use GetDICOMImportJob.

SDK for JavaScript (v3)

```

import { GetDICOMImportJobCommand } from "@aws-sdk/client-medical-imaging";
import { medicalImagingClient } from "../libs/medicalImagingClient.js";

/**
 * @param {string} datastoreId - The ID of the data store.
 * @param {string} jobId - The ID of the import job.
 */
export const getDICOMImportJob = async (
  datastoreId = "xxxxxxxxxxxxxxxxxxxxxxxx",
  jobId = "xxxxxxxxxxxxxxxxxxxxxxxx"
) => {
  const response = await medicalImagingClient.send(
    new GetDICOMImportJobCommand({ datastoreId: datastoreId, jobId: jobId })
  );
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: 'a2637936-78ea-44e7-98b8-7a87d95dfaee',
  //     extendedRequestId: undefined,

```

```

//      cfId: undefined,
//      attempts: 1,
//      totalRetryDelay: 0
// },
//   jobProperties: {
//     dataAccessRoleArn: 'arn:aws:iam:xxxxxxxxxxxx:role/dicom_import',
//     datastoreId: 'xxxxxxxxxxxxxxxxxxxxxxxxxxxx',
//     endedAt: 2023-09-19T17:29:21.753Z,
//     inputS3Uri: 's3://healthimaging-source/CTStudy/',
//     jobId: 'xxxxxxxxxxxxxxxxxxxxxxxxxxxx',
//     jobName: 'job_1',
//     jobStatus: 'COMPLETED',
//     outputS3Uri: 's3://health-imaging-dest/
ouput_ct/'xxxxxxxxxxxxxxxxxxxxxxxxxxxx'-DicomImport-'xxxxxxxxxxxxxxxxxxxxxxxxxxxx'/',
//     submittedAt: 2023-09-19T17:27:25.143Z
//   }
// }

return response;
};

```

- For API details, see [GetDICOMImportJob](#) in *AWS SDK for JavaScript API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

GetDatastore

The following code example shows how to use GetDatastore.

SDK for JavaScript (v3)

```

import { GetDatastoreCommand } from "@aws-sdk/client-medical-imaging";
import { medicalImagingClient } from "../libs/medicalImagingClient.js";

/**
 * @param {string} datastoreID - The ID of the data store.
 */
export const getDatastore = async (datastoreID = "DATASTORE_ID") => {

```

```
const response = await medicalImagingClient.send(
  new GetDatastoreCommand({ datastoreId: datastoreID })
);
console.log(response);
// {
//   '$metadata': {
//     httpStatusCode: 200,
//     requestId: '55ea7d2e-222c-4a6a-871e-4f591f40cadb',
//     extendedRequestId: undefined,
//     cfId: undefined,
//     attempts: 1,
//     totalRetryDelay: 0
//   },
//   datastoreProperties: {
//     createdAt: 2023-08-04T18:50:36.239Z,
//     datastoreArn: 'arn:aws:medical-imaging:us-east-1:xxxxxxxxx:datastore/
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',
//     datastoreId: 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',
//     datastoreName: 'my_datastore',
//     datastoreStatus: 'ACTIVE',
//     updatedAt: 2023-08-04T18:50:36.239Z
//   }
// }
return response["datastoreProperties"];
};
```

- For API details, see [GetDatastore](#) in *AWS SDK for JavaScript API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

GetImageFrame

The following code example shows how to use `GetImageFrame`.

SDK for JavaScript (v3)

```
import { GetImageFrameCommand } from "@aws-sdk/client-medical-imaging";
```

```
import { medicalImagingClient } from "../libs/medicalImagingClient.js";

/**
 * @param {string} imageFrameFileName - The name of the file for the HTJ2K-encoded
 image frame.
 * @param {string} datastoreId - The data store's ID.
 * @param {string} imageSetID - The image set's ID.
 * @param {string} imageFrameID - The image frame's ID.
 */
export const getImageFrame = async (
  imageFrameFileName = "image.jph",
  datastoreID = "DATASTORE_ID",
  imageSetID = "IMAGE_SET_ID",
  imageFrameID = "IMAGE_FRAME_ID"
) => {
  const response = await medicalImagingClient.send(
    new GetImageFrameCommand({
      datastoreId: datastoreID,
      imageSetId: imageSetID,
      imageFrameInformation: { imageFrameId: imageFrameID },
    })
  );
  const buffer = await response.imageFrameBlob.transformToByteArray();
  writeFileSync(imageFrameFileName, buffer);

  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: 'e4ab42a5-25a3-4377-873f-374ecf4380e1',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  //   contentType: 'application/octet-stream',
  //   imageFrameBlob: <ref *1> IncomingMessage {}
  // }
  return response;
};
```

- For API details, see [GetImageFrame](#) in *AWS SDK for JavaScript API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

GetImageSet

The following code example shows how to use GetImageSet.

SDK for JavaScript (v3)

```
import { GetImageSetCommand } from "@aws-sdk/client-medical-imaging";
import { medicalImagingClient } from "../libs/medicalImagingClient.js";

/**
 * @param {string} datastoreId - The ID of the data store.
 * @param {string} imageSetId - The ID of the image set.
 * @param {string} imageSetVersion - The optional version of the image set.
 */
export const getImageSet = async (
  datastoreId = "xxxxxxxxxxxxxxxx",
  imageSetId = "xxxxxxxxxxxxxxxx",
  imageSetVersion = ""
) => {
  let params = { datastoreId: datastoreId, imageSetId: imageSetId };
  if (imageSetVersion !== "") {
    params.imageSetVersion = imageSetVersion;
  }
  const response = await medicalImagingClient.send(
    new GetImageSetCommand(params)
  );
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: '0615c161-410d-4d06-9d8c-6e1241bb0a5a',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  // }
```



```
//   createdAt: 2023-09-22T14:49:26.427Z,  
//   datastoreId: 'xxxxxxxxxxxxxxxxx',  
//   imageSetArn: 'arn:aws:medical-imaging:us-east-1:xxxxxxxxxx:datastore/  
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx/imagetset/xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',  
//   imageSetId: 'xxxxxxxxxxxxxxxxx',  
//   imageSetState: 'ACTIVE',  
//   imageSetWorkflowStatus: 'CREATED',  
//   updatedAt: 2023-09-22T14:49:26.427Z,  
//   versionId: '1'  
// }  
  
return response;  
};
```

- For API details, see [GetImageSet](#) in *AWS SDK for JavaScript API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

GetImageSetMetadata

The following code example shows how to use `GetImageSetMetadata`.

SDK for JavaScript (v3)

Utility function to get image set metadata.

```
import { GetImageSetMetadataCommand } from "@aws-sdk/client-medical-imaging";  
import { medicalImagingClient } from "../libs/medicalImagingClient.js";  
import { writeFileSync } from "fs";  
  
/**  
 * @param {string} metadataFileName - The name of the file for the gzipped metadata.  
 * @param {string} datastoreId - The ID of the data store.  
 * @param {string} imageSetId - The ID of the image set.  
 * @param {string} versionID - The optional version ID of the image set.  
 */
```

```
export const getImageSetMetadata = async (
  metadataFileName = "metadata.json.gzip",
  datastoreId = "xxxxxxxxxxxxxxxx",
  imageSetId = "xxxxxxxxxxxxxxxx",
  versionID = ""
) => {
  const params = { datastoreId: datastoreId, imageSetId: imageSetId };

  if (versionID) {
    params.versionID = versionID;
  }

  const response = await medicalImagingClient.send(
    new GetImageSetMetadataCommand(params)
  );
  const buffer = await response.imageSetMetadataBlob.transformToByteArray();
  writeFileSync(metadataFileName, buffer);

  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: '5219b274-30ff-4986-8cab-48753de3a599',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  //   contentType: 'application/json',
  //   contentEncoding: 'gzip',
  //   imageSetMetadataBlob: <ref *1> IncomingMessage {}
  // }

  return response;
};
```

Get image set metadata without version.

```
try {
  await getImageSetMetadata(
    "metadata.json.gzip",
```

```
    "12345678901234567890123456789012",
    "12345678901234567890123456789012"
  );
} catch (err) {
  console.log("Error", err);
}
```

Get image set metadata with version.

```
try {
  await getImageSetMetadata(
    "metadata2.json.gzip",
    "12345678901234567890123456789012",
    "12345678901234567890123456789012",
    "1"
  );
} catch (err) {
  console.log("Error", err);
}
```

- For API details, see [GetImageSetMetadata](#) in *AWS SDK for JavaScript API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

ListDICOMImportJobs

The following code example shows how to use `ListDICOMImportJobs`.

SDK for JavaScript (v3)

```
import { paginateListDICOMImportJobs } from "@aws-sdk/client-medical-imaging";
import { medicalImagingClient } from "../libs/medicalImagingClient.js";

/**
 * @param {string} datastoreId - The ID of the data store.
 */
```

```

export const listDICOMImportJobs = async (
  datastoreId = "xxxxxxxxxxxxxxxxxxxxx"
) => {
  const paginatorConfig = {
    client: medicalImagingClient,
    pageSize: 50,
  };

  const commandParams = { datastoreId: datastoreId };
  const paginator = paginateListDICOMImportJobs(paginatorConfig, commandParams);

  let jobSummaries = [];
  for await (const page of paginator) {
    // Each page contains a list of `jobSummaries`. The list is truncated if is
    // larger than `pageSize`.
    jobSummaries.push(...page["jobSummaries"]);
    console.log(page);
  }
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: '3c20c66e-0797-446a-a1d8-91b742fd15a0',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  //   jobSummaries: [
  //     {
  //       dataAccessRoleArn: 'arn:aws:iam:xxxxxxxxxxxx:role/dicom_import',
  //       datastoreId: 'xxxxxxxxxxxxxxxxxxxxx',
  //       endedAt: 2023-09-22T14:49:51.351Z,
  //       jobId: 'xxxxxxxxxxxxxxxxxxxxx',
  //       jobName: 'test-1',
  //       jobStatus: 'COMPLETED',
  //       submittedAt: 2023-09-22T14:48:45.767Z
  //     }
  //   ]
  // }

  return jobSummaries;
};

```

- For API details, see [ListDICOMImportJobs](#) in *AWS SDK for JavaScript API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

ListDatastores

The following code example shows how to use `ListDatastores`.

SDK for JavaScript (v3)

```
import { paginateListDatastores } from "@aws-sdk/client-medical-imaging";
import { medicalImagingClient } from "../libs/medicalImagingClient.js";

export const listDatastores = async () => {
  const paginatorConfig = {
    client: medicalImagingClient,
    pageSize: 50,
  };

  const commandParams = {};
  const paginator = paginateListDatastores(paginatorConfig, commandParams);

  /**
   * @type {import("@aws-sdk/client-medical-imaging").DatastoreSummary[]}
   */
  const datastoreSummaries = [];
  for await (const page of paginator) {
    // Each page contains a list of `jobSummaries`. The list is truncated if is
    larger than `pageSize`.
    datastoreSummaries.push(...page["datastoreSummaries"]);
    console.log(page);
  }
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: '6aa99231-d9c2-4716-a46e-edb830116fa3',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },

```

```

// datastoreSummaries: [
//   {
//     createdAt: 2023-08-04T18:49:54.429Z,
//     datastoreArn: 'arn:aws:medical-imaging:us-east-1:xxxxxxxxx:datastore/
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',
//     datastoreId: 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',
//     datastoreName: 'my_datastore',
//     datastoreStatus: 'ACTIVE',
//     updatedAt: 2023-08-04T18:49:54.429Z
//   }
//   ...
// ]
// }

return datastoreSummaries;
};

```

- For API details, see [ListDatastores](#) in *AWS SDK for JavaScript API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

ListImageSetVersions

The following code example shows how to use ListImageSetVersions.

SDK for JavaScript (v3)

```

import { paginateListImageSetVersions } from "@aws-sdk/client-medical-imaging";
import { medicalImagingClient } from "../libs/medicalImagingClient.js";

/**
 * @param {string} datastoreId - The ID of the data store.
 * @param {string} imageSetId - The ID of the image set.
 */
export const listImageSetVersions = async (
  datastoreId = "xxxxxxxxxxxxx",
  imageSetId = "xxxxxxxxxxxxx"

```

```
) => {
  const paginatorConfig = {
    client: medicalImagingClient,
    pageSize: 50,
  };

  const commandParams = { datastoreId, imageSetId };
  const paginator = paginateListImageSetVersions(
    paginatorConfig,
    commandParams
  );

  let imageSetPropertiesList = [];
  for await (const page of paginator) {
    // Each page contains a list of `jobSummaries`. The list is truncated if is
    // larger than `pageSize`.
    imageSetPropertiesList.push(...page["imageSetPropertiesList"]);
    console.log(page);
  }
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: '74590b37-a002-4827-83f2-3c590279c742',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  //   imageSetPropertiesList: [
  //     {
  //       ImageSetWorkflowStatus: 'CREATED',
  //       createdAt: 2023-09-22T14:49:26.427Z,
  //       imageSetId: 'xxxxxxxxxxxxxxxxxxxxxxxxxxxx',
  //       imageSetState: 'ACTIVE',
  //       versionId: '1'
  //     }
  //   ]
  // }
  return imageSetPropertiesList;
};
```

- For API details, see [ListImageSetVersions](#) in *AWS SDK for JavaScript API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

ListTagsForResource

The following code example shows how to use ListTagsForResource.

SDK for JavaScript (v3)

```
import { ListTagsForResourceCommand } from "@aws-sdk/client-medical-imaging";
import { medicalImagingClient } from "../libs/medicalImagingClient.js";

/**
 * @param {string} resourceArn - The Amazon Resource Name (ARN) for the data store
 * or image set.
 */
export const listTagsForResource = async (
  resourceArn = "arn:aws:medical-imaging:us-east-1:abc:datastore/def/imageset/ghi"
) => {
  const response = await medicalImagingClient.send(
    new ListTagsForResourceCommand({ resourceArn: resourceArn })
  );
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: '008fc6d3-abec-4870-a155-20fa3631e645',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  //   tags: { Deployment: 'Development' }
  // }

  return response;
};
```


- For API details, see [ListTagsForResource](#) in *AWS SDK for JavaScript API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

SearchImageSets

The following code example shows how to use SearchImageSets.

SDK for JavaScript (v3)

The utility function for searching image sets.

```
import {paginateSearchImageSets} from "@aws-sdk/client-medical-imaging";
import {medicalImagingClient} from "../libs/medicalImagingClient.js";

/**
 * @param {string} datastoreId - The data store's ID.
 * @param { import('@aws-sdk/client-medical-imaging').SearchFilter[] } filters - The
search criteria filters.
 * @param { import('@aws-sdk/client-medical-imaging').Sort } sort - The search
criteria sort.
 */
export const searchImageSets = async (
  datastoreId = "xxxxxxxx",
  searchCriteria = {}
) => {
  const paginatorConfig = {
    client: medicalImagingClient,
    pageSize: 50,
  };

  const commandParams = {
    datastoreId: datastoreId,
    searchCriteria: searchCriteria,
  };

  const paginator = paginateSearchImageSets(paginatorConfig, commandParams);

  const imageSetsMetadataSummaries = [];
```

```
    for await (const page of paginator) {
      // Each page contains a list of `jobSummaries`. The list is truncated if is
larger than `pageSize`.
      imageSetsMetadataSummaries.push(...page["imageSetsMetadataSummaries"]);
      console.log(page);
    }
    // {
    //   '$metadata': {
    //     httpStatusCode: 200,
    //     requestId: 'f009ea9c-84ca-4749-b5b6-7164f00a5ada',
    //     extendedRequestId: undefined,
    //     cfId: undefined,
    //     attempts: 1,
    //     totalRetryDelay: 0
    //   },
    //   imageSetsMetadataSummaries: [
    //     {
    //       DICOMTags: [Object],
    //       createdAt: "2023-09-19T16:59:40.551Z",
    //       imageSetId: '7f75e1b5c0f40eac2b24cf712f485f50',
    //       updatedAt: "2023-09-19T16:59:40.551Z",
    //       version: 1
    //     }
    //   ]
    // }

    return imageSetsMetadataSummaries;
  };
```

Use case #1: EQUAL operator.

```
const datastoreId = "12345678901234567890123456789012";

try {
  const searchCriteria = {
    filters: [
      {
        values: [{DICOMPatientId: "1234567"}],
        operator: "EQUAL",
      },
    ],
  };
```

```
    await searchImageSets(datastoreId, searchCriteria);
  } catch (err) {
    console.error(err);
  }
```

Use case #2: BETWEEN operator using DICOMStudyDate and DICOMStudyTime.

```
const datastoreId = "12345678901234567890123456789012";

try {
  const searchCriteria = {
    filters: [
      {
        values: [
          {
            DICOMStudyDateAndTime: {
              DICOMStudyDate: "19900101",
              DICOMStudyTime: "000000",
            },
          },
          {
            DICOMStudyDateAndTime: {
              DICOMStudyDate: "20230901",
              DICOMStudyTime: "000000",
            },
          },
        ],
        operator: "BETWEEN",
      },
    ],
  };

  await searchImageSets(datastoreId, searchCriteria);
} catch (err) {
  console.error(err);
}
```

Use case #3: BETWEEN operator using createdAt. Time studies were previously persisted.

```
const datastoreId = "12345678901234567890123456789012";
```

```
try {
  const searchCriteria = {
    filters: [
      {
        values: [
          {createdAt: new Date("1985-04-12T23:20:50.52Z")},
          {createdAt: new Date()},
        ],
        operator: "BETWEEN",
      },
    ],
  };

  await searchImageSets(datastoreId, searchCriteria);
} catch (err) {
  console.error(err);
}
```

Use case #4: EQUAL operator on DICOMSeriesInstanceUID and BETWEEN on updatedAt and sort response in ASC order on updatedAt field.

```
const datastoreId = "12345678901234567890123456789012";

try {
  const searchCriteria = {
    filters: [
      {
        values: [
          {updatedAt: new Date("1985-04-12T23:20:50.52Z")},
          {updatedAt: new Date()},
        ],
        operator: "BETWEEN",
      },
      {
        values: [
          {DICOMSeriesInstanceUID:
"1.1.123.123456.1.12.1.1234567890.1234.12345678.123"},
        ],
        operator: "EQUAL",
      },
    ],
    sort: {
```

```
        sortOrder: "ASC",
        sortField: "updatedAt",
    }
};

    await searchImageSets(datastoreId, searchCriteria);
} catch (err) {
    console.error(err);
}
```

- For API details, see [SearchImageSets](#) in *AWS SDK for JavaScript API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

StartDICOMImportJob

The following code example shows how to use `StartDICOMImportJob`.

SDK for JavaScript (v3)

```
import { StartDICOMImportJobCommand } from "@aws-sdk/client-medical-imaging";
import { medicalImagingClient } from "../libs/medicalImagingClient.js";

/**
 * @param {string} jobName - The name of the import job.
 * @param {string} datastoreId - The ID of the data store.
 * @param {string} dataAccessRoleArn - The Amazon Resource Name (ARN) of the role
that grants permission.
 * @param {string} inputS3Uri - The URI of the S3 bucket containing the input files.
 * @param {string} outputS3Uri - The URI of the S3 bucket where the output files are
stored.
 */
export const startDicomImportJob = async (
    jobName = "test-1",
    datastoreId = "12345678901234567890123456789012",
    dataAccessRoleArn = "arn:aws:iam::xxxxxxxxxxxx:role/ImportJobDataAccessRole",
    inputS3Uri = "s3://medical-imaging-dicom-input/dicom_input/",
```

```
outputS3Uri = "s3://medical-imaging-output/job_output/"
) => {
  const response = await medicalImagingClient.send(
    new StartDICOMImportJobCommand({
      jobName: jobName,
      datastoreId: datastoreId,
      dataAccessRoleArn: dataAccessRoleArn,
      inputS3Uri: inputS3Uri,
      outputS3Uri: outputS3Uri,
    })
  );
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: '6e81d191-d46b-4e48-a08a-cdcc7e11eb79',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  //   datastoreId: 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',
  //   jobId: 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',
  //   jobStatus: 'SUBMITTED',
  //   submittedAt: 2023-09-22T14:48:45.767Z
  // }
  return response;
};
```

- For API details, see [StartDICOMImportJob](#) in *AWS SDK for JavaScript API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

TagResource

The following code example shows how to use TagResource.

SDK for JavaScript (v3)

```
import { TagResourceCommand } from "@aws-sdk/client-medical-imaging";
import { medicalImagingClient } from "../libs/medicalImagingClient.js";

/**
 * @param {string} resourceArn - The Amazon Resource Name (ARN) for the data store
 * or image set.
 * @param {Record<string,string>} tags - The tags to add to the resource as JSON.
 * - For example: {"Deployment" : "Development"}
 */
export const tagResource = async (
  resourceArn = "arn:aws:medical-imaging:us-east-1:xxxxxx:datastore/xxxxx/imageset/
  xxx",
  tags = {}
) => {
  const response = await medicalImagingClient.send(
    new TagResourceCommand({ resourceArn: resourceArn, tags: tags })
  );
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 204,
  //     requestId: '8a6de9a3-ec8e-47ef-8643-473518b19d45',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   }
  // }

  return response;
};
```

- For API details, see [TagResource](#) in *AWS SDK for JavaScript API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

UntagResource

The following code example shows how to use UntagResource.

SDK for JavaScript (v3)

```
import { UntagResourceCommand } from "@aws-sdk/client-medical-imaging";
import { medicalImagingClient } from "../libs/medicalImagingClient.js";

/**
 * @param {string} resourceArn - The Amazon Resource Name (ARN) for the data store
 * or image set.
 * @param {string[]} tagKeys - The keys of the tags to remove.
 */
export const untagResource = async (
  resourceArn = "arn:aws:medical-imaging:us-east-1:xxxxxx:datastore/xxxxx/imageset/
xxx",
  tagKeys = []
) => {
  const response = await medicalImagingClient.send(
    new UntagResourceCommand({ resourceArn: resourceArn, tagKeys: tagKeys })
  );
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 204,
  //     requestId: '8a6de9a3-ec8e-47ef-8643-473518b19d45',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   }
  // }

  return response;
};
```

- For API details, see [UntagResource](#) in *AWS SDK for JavaScript API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

UpdateImageSetMetadata

The following code example shows how to use UpdateImageSetMetadata.

SDK for JavaScript (v3)

```
import {UpdateImageSetMetadataCommand} from "@aws-sdk/client-medical-imaging";
import {medicalImagingClient} from "../libs/medicalImagingClient.js";

/**
 * @param {string} datastoreId - The ID of the HealthImaging data store.
 * @param {string} imageSetId - The ID of the HealthImaging image set.
 * @param {string} latestVersionId - The ID of the HealthImaging image set version.
 * @param {{}} updateMetadata - The metadata to update.
 * @param {boolean} force - Force the update.
 */
export const updateImageSetMetadata = async (datastoreId = "xxxxxxxxxx",
                                             imageSetId = "xxxxxxxxxx",
                                             latestVersionId = "1",
                                             updateMetadata = '{}',
                                             force = false) => {

  try {
    const response = await medicalImagingClient.send(
      new UpdateImageSetMetadataCommand({
        datastoreId: datastoreId,
        imageSetId: imageSetId,
        latestVersionId: latestVersionId,
        updateImageSetMetadataUpdates: updateMetadata,
        force: force,
      })
    );
    console.log(response);
    // {
    //   '$metadata': {
    //     httpStatusCode: 200,
    //     requestId: '7966e869-e311-4bff-92ec-56a61d3003ea',
    //     extendedRequestId: undefined,
```

```
        //      cfId: undefined,
        //      attempts: 1,
        //      totalRetryDelay: 0
    // },
    //      createdAt: 2023-09-22T14:49:26.427Z,
    //      datastoreId: 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',
    //      imageSetId: 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',
    //      imageSetState: 'LOCKED',
    //      imageSetWorkflowStatus: 'UPDATING',
    //      latestVersionId: '4',
    //      updatedAt: 2023-09-27T19:41:43.494Z
    // }
    return response;
} catch (err) {
    console.error(err);
}
};
```

Use case #1: Insert or update an attribute and force the update.

```
const insertAttributes =
    JSON.stringify({
        "SchemaVersion": 1.1,
        "Study": {
            "DICOM": {
                "StudyDescription": "CT CHEST"
            }
        }
    });

const updateMetadata = {
    "DICOMUpdates": {
        "updatableAttributes":
            new TextEncoder().encode(insertAttributes)
    }
};

await updateImageSetMetadata(datastoreId, imageSetID,
    versionID, updateMetadata, true);
```

Use case #2: Remove an attribute.

```
// Attribute key and value must match the existing attribute.
const remove_attribute =
  JSON.stringify({
    "SchemaVersion": 1.1,
    "Study": {
      "DICOM": {
        "StudyDescription": "CT CHEST"
      }
    }
  });

const updateMetadata = {
  "DICOMUpdates": {
    "removableAttributes":
      new TextEncoder().encode(remove_attribute)
  }
};

await updateImageSetMetadata(datastoreID, imageSetID,
  versionID, updateMetadata);
```

Use case #3: Remove an instance.

```
const remove_instance =
  JSON.stringify({
    "SchemaVersion": 1.1,
    "Study": {
      "Series": {
        "1.1.1.1.1.1.12345.123456789012.123.12345678901234.1": {
          "Instances": {

"1.1.1.1.1.1.12345.123456789012.123.12345678901234.1": {}

          }
        }
      }
    }
  });

const updateMetadata = {
  "DICOMUpdates": {
    "removableAttributes":
      new TextEncoder().encode(remove_instance)
```

```
    }  
  };  
  
  await updateImageSetMetadata(datastoreId, imageSetID,  
    versionID, updateMetadata);
```

Use case #4: Revert to an earlier version.

```
const updateMetadata = {  
  "revertToVersionId": "1"  
};  
  
await updateImageSetMetadata(datastoreId, imageSetID,  
  versionID, updateMetadata);
```

- For API details, see [UpdateImageSetMetadata](#) in *AWS SDK for JavaScript API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Scenarios

Get started with image sets and image frames

The following code example shows how to import DICOM files and download image frames in HealthImaging.

The implementation is structured as a workflow command-line application.

- Set up resources for a DICOM import.
- Import DICOM files into a data store.
- Retrieve the image set IDs for the import job.
- Retrieve the image frame IDs for the image sets.
- Download, decode and verify the image frames.
- Clean up resources.

SDK for JavaScript (v3)

index.js - Orchestrate steps.

```
import {
  parseScenarioArgs,
  Scenario,
} from "@aws-doc-sdk-examples/lib/scenario/index.js";
import {
  saveState,
  loadState,
} from "@aws-doc-sdk-examples/lib/scenario/steps-common.js";

import {
  createStack,
  deployStack,
  getAccountId,
  getDatastoreName,
  getStackName,
  outputState,
  waitForStackCreation,
} from "../deploy-steps.js";
import {
  doCopy,
  selectDataset,
  copyDataset,
  outputCopiedObjects,
} from "../dataset-steps.js";
import {
  doImport,
  outputImportJobStatus,
  startDICOMImport,
  waitForImportJobCompletion,
} from "../import-steps.js";
import {
  getManifestFile,
  outputImageSetIds,
  parseManifestFile,
} from "../image-set-steps.js";
import {
  getImageSetMetadata,
  outputImageFrameIds,
} from "../image-frame-steps.js";
```

```
import { decodeAndVerifyImages, doVerify } from "./verify-steps.js";
import {
  confirmCleanup,
  deleteImageSets,
  deleteStack,
} from "./clean-up-steps.js";

const context = {};

const scenarios = {
  deploy: new Scenario(
    "Deploy Resources",
    [
      deployStack,
      getStackName,
      getDatastoreName,
      getAccountId,
      createStack,
      waitForStackCreation,
      outputState,
      saveState,
    ],
    context,
  ),
  demo: new Scenario(
    "Run Demo",
    [
      loadState,
      doCopy,
      selectDataset,
      copyDataset,
      outputCopiedObjects,
      doImport,
      startDICOMImport,
      waitForImportJobCompletion,
      outputImportJobStatus,
      getManifestFile,
      parseManifestFile,
      outputImageSetIds,
      getImageSetMetadata,
      outputImageFrameIds,
      doVerify,
      decodeAndVerifyImages,
      saveState,
    ],
  ),
};
```

```
    ],
    context,
  ),
  destroy: new Scenario(
    "Clean Up Resources",
    [loadState, confirmCleanup, deleteImageSets, deleteStack],
    context,
  ),
};

// Call function if run directly
import { fileURLToPath } from "url";
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  parseScenarioArgs(scenarios);
}
```

deploy-steps.js - Deploy resources.

```
import fs from "node:fs/promises";
import path from "node:path";

import {
  CloudFormationClient,
  CreateStackCommand,
  DescribeStacksCommand,
} from "@aws-sdk/client-cloudformation";
import { STSClient, GetCallerIdentityCommand } from "@aws-sdk/client-sts";

import {
  ScenarioAction,
  ScenarioInput,
  ScenarioOutput,
} from "@aws-doc-sdk-examples/lib/scenario/index.js";
import { retry } from "@aws-doc-sdk-examples/lib/utils/util-timers.js";

const cfnClient = new CloudFormationClient({});
const stsClient = new STSClient({});

const __dirname = path.dirname(new URL(import.meta.url).pathname);
const cfnTemplatePath = path.join(
  __dirname,
  "../../../../../workflows/healthimaging_image_sets/resources/cfn_template.yaml",

```

```
);

export const deployStack = new ScenarioInput(
  "deployStack",
  "Do you want to deploy the CloudFormation stack?",
  { type: "confirm" },
);

export const getStackName = new ScenarioInput(
  "getStackName",
  "Enter a name for the CloudFormation stack:",
  { type: "input", skipWhen: (/** @type {} */ state) => !state.deployStack },
);

export const getDatastoreName = new ScenarioInput(
  "getDatastoreName",
  "Enter a name for the HealthImaging datastore:",
  { type: "input", skipWhen: (/** @type {} */ state) => !state.deployStack },
);

export const getAccountId = new ScenarioAction(
  "getAccountId",
  async (/** @type {} */ state) => {
    const command = new GetCallerIdentityCommand({});
    const response = await stsClient.send(command);
    state.accountId = response.Account;
  },
  {
    skipWhen: (/** @type {} */ state) => !state.deployStack,
  },
);

export const createStack = new ScenarioAction(
  "createStack",
  async (/** @type {} */ state) => {
    const stackName = state.getStackName;
    const datastoreName = state.getDatastoreName;
    const accountId = state.accountId;

    const command = new CreateStackCommand({
      StackName: stackName,
      TemplateBody: await fs.readFile(cfnTemplatePath, "utf8"),
      Capabilities: ["CAPABILITY_IAM"],
      Parameters: [
```



```

    {
      ParameterKey: "datastoreName",
      ParameterValue: datastoreName,
    },
    {
      ParameterKey: "userAccountID",
      ParameterValue: accountId,
    },
  ],
});

const response = await cfnClient.send(command);
state.stackId = response.StackId;
},
{ skipWhen: (/** @type {} */ state) => !state.deployStack },
);

export const waitForStackCreation = new ScenarioAction(
  "waitForStackCreation",
  async (/** @type {} */ state) => {
    const command = new DescribeStacksCommand({
      StackName: state.stackId,
    });

    await retry({ intervalInMs: 10000, maxRetries: 60 }, async () => {
      const response = await cfnClient.send(command);
      const stack = response.Stacks?.find(
        (s) => s.StackName == state.getStackName,
      );
      if (!stack || stack.StackStatus === "CREATE_IN_PROGRESS") {
        throw new Error("Stack creation is still in progress");
      }
      if (stack.StackStatus === "CREATE_COMPLETE") {
        state.stackOutputs = stack.Outputs?.reduce((acc, output) => {
          acc[output.OutputKey] = output.OutputValue;
          return acc;
        }, {});
      } else {
        throw new Error(
          `Stack creation failed with status: ${stack.StackStatus}`,
        );
      }
    });
  });
},
);

```

```

    {
      skipWhen: (/** @type {} */ state) => !state.deployStack,
    },
  );

export const outputState = new ScenarioOutput(
  "outputState",
  (/** @type {} */ state) => {
    /**
     * @type {{ stackOutputs: { DatastoreID: string, BucketName: string, RoleArn:
string }}}
     */
    const { stackOutputs } = state;
    return `Stack creation completed. Output values:
Datastore ID: ${stackOutputs?.DatastoreID}
Bucket Name: ${stackOutputs?.BucketName}
Role ARN: ${stackOutputs?.RoleArn}
`;
  },
  { skipWhen: (/** @type {} */ state) => !state.deployStack },
);

```

dataset-steps.js - Copy DICOM files.

```

import {
  S3Client,
  CopyObjectCommand,
  ListObjectsV2Command,
} from "@aws-sdk/client-s3";

import {
  ScenarioAction,
  ScenarioInput,
  ScenarioOutput,
} from "@aws-doc-sdk-examples/lib/scenario/index.js";

const s3Client = new S3Client({});

const datasetOptions = [
  {
    name: "CT of chest (2 images)",
    value: "00029d25-fb18-4d42-aaa5-a0897d1ac8f7",
  },
];

```

```
    },
    {
      name: "CT of pelvis (57 images)",
      value: "00025d30-ef8f-4135-a35a-d83eff264fc1",
    },
    {
      name: "MRI of head (192 images)",
      value: "0002d261-8a5d-4e63-8e2e-0cbfac87b904",
    },
    {
      name: "MRI of breast (92 images)",
      value: "0002dd07-0b7f-4a68-a655-44461ca34096",
    },
  ],
];

/**
 * @typedef {{ stackOutputs: {
 *   BucketName: string,
 *   DatastoreID: string,
 *   doCopy: boolean
 * }}} State
 */

export const selectDataset = new ScenarioInput(
  "selectDataset",
  (state) => {
    if (!state.doCopy) {
      process.exit(0);
    }
    return "Select a DICOM dataset to import:";
  },
  {
    type: "select",
    choices: datasetOptions,
  },
);

export const doCopy = new ScenarioInput(
  "doCopy",
  "Do you want to copy images from the public dataset into your bucket?",
  {
    type: "confirm",
  },
);
```

```
export const copyDataset = new ScenarioAction(
  "copyDataset",
  async (** @type { State } */ state) => {
    const inputBucket = state.stackOutputs.BucketName;
    const inputPrefix = `input/`;
    const selectedDatasetId = state.selectDataset;

    const sourceBucket = "idc-open-data";
    const sourcePrefix = `${selectedDatasetId}`;

    const listObjectsCommand = new ListObjectsV2Command({
      Bucket: sourceBucket,
      Prefix: sourcePrefix,
    });

    const objects = await s3Client.send(listObjectsCommand);

    const copyPromises = objects.Contents.map((object) => {
      const sourceKey = object.Key;
      const destinationKey = `${inputPrefix}${sourceKey}
        .split("/")
        .slice(1)
        .join("/")}`;

      const copyCommand = new CopyObjectCommand({
        Bucket: inputBucket,
        CopySource: `${sourceBucket}/${sourceKey}`,
        Key: destinationKey,
      });

      return s3Client.send(copyCommand);
    });

    const results = await Promise.all(copyPromises);
    state.copiedObjects = results.length;
  },
);

export const outputCopiedObjects = new ScenarioOutput(
  "outputCopiedObjects",
  (state) => `${state.copiedObjects} DICOM files were copied.`
);
```

import-steps.js - Start import into datastore.

```
import {
  MedicalImagingClient,
  StartDICOMImportJobCommand,
  GetDICOMImportJobCommand,
} from "@aws-sdk/client-medical-imaging";

import {
  ScenarioAction,
  ScenarioOutput,
  ScenarioInput,
} from "@aws-doc-sdk-examples/lib/scenario/index.js";
import { retry } from "@aws-doc-sdk-examples/lib/utils/util-timers.js";

/**
 * @typedef {{ stackOutputs: {
 *   BucketName: string,
 *   DatastoreID: string,
 *   RoleArn: string
 * }}} State
 */

export const doImport = new ScenarioInput(
  "doImport",
  "Do you want to import DICOM images into your datastore?",
  {
    type: "confirm",
    default: true,
  },
);

export const startDICOMImport = new ScenarioAction(
  "startDICOMImport",
  async (** @type {State} */ state) => {
    if (!state.doImport) {
      process.exit(0);
    }
    const medicalImagingClient = new MedicalImagingClient({});
    const inputS3Uri = `s3://${state.stackOutputs.BucketName}/input/`;
    const outputS3Uri = `s3://${state.stackOutputs.BucketName}/output/`;
```

```
const command = new StartDICOMImportJobCommand({
  dataAccessRoleArn: state.stackOutputs.RoleArn,
  datastoreId: state.stackOutputs.DatastoreID,
  inputS3Uri,
  outputS3Uri,
});

const response = await medicalImagingClient.send(command);
state.importJobId = response.jobId;
},
);

export const waitForImportJobCompletion = new ScenarioAction(
  "waitForImportJobCompletion",
  async (** @type {State} */ state) => {
    const medicalImagingClient = new MedicalImagingClient({});
    const command = new GetDICOMImportJobCommand({
      datastoreId: state.stackOutputs.DatastoreID,
      jobId: state.importJobId,
    });

    await retry({ intervalInMs: 10000, maxRetries: 60 }, async () => {
      const response = await medicalImagingClient.send(command);
      const jobStatus = response.jobProperties?.jobStatus;
      if (!jobStatus || jobStatus === "IN_PROGRESS") {
        throw new Error("Import job is still in progress");
      }
      if (jobStatus === "COMPLETED") {
        state.importJobOutputS3Uri = response.jobProperties.outputS3Uri;
      } else {
        throw new Error(`Import job failed with status: ${jobStatus}`);
      }
    });
  },
);

export const outputImportJobStatus = new ScenarioOutput(
  "outputImportJobStatus",
  (state) =>
    `DICOM import job completed. Output location: ${state.importJobOutputS3Uri}`,
);
```

image-set-steps.js - Get image set IDs.

```
import { S3Client, GetObjectCommand } from "@aws-sdk/client-s3";

import {
  ScenarioAction,
  ScenarioOutput,
} from "@aws-doc-sdk-examples/lib/scenario/index.js";

/**
 * @typedef {{ stackOutputs: {
 *   BucketName: string,
 *   DatastoreID: string,
 *   RoleArn: string
 * }, importJobId: string,
 * importJobOutputS3Uri: string,
 * imageSetIds: string[],
 * manifestContent: { jobSummary: { imageSetsSummary: { imageSetId: string }[] } }
 * }} State
 */

const s3Client = new S3Client({});

export const getManifestFile = new ScenarioAction(
  "getManifestFile",
  async (/** @type {State} */ state) => {
    const bucket = state.stackOutputs.BucketName;
    const prefix = `output/${state.stackOutputs.DatastoreID}-DicomImport-
${state.importJobId}/`;
    const key = `${prefix}job-output-manifest.json`;

    const command = new GetObjectCommand({
      Bucket: bucket,
      Key: key,
    });

    const response = await s3Client.send(command);
    const manifestContent = await response.Body.transformToString();
    state.manifestContent = JSON.parse(manifestContent);
  },
);

export const parseManifestFile = new ScenarioAction(
  "parseManifestFile",
```

```

(** @type {State} */ state) => {
  const imageSetIds =
    state.manifestContent.jobSummary.imageSetsSummary.reduce(
      (imageSetIds, next) => {
        return { ...imageSetIds, [next.imageSetId]: next.imageSetId };
      },
      {},
    );
  state.imageSetIds = Object.keys(imageSetIds);
},
);

export const outputImageSetIds = new ScenarioOutput(
  "outputImageSetIds",
  (** @type {State} */ state) =>
    `The image sets created by this import job are: \n${state.imageSetIds
      .map((id) => `Image set: ${id}`)
      .join("\n")}` ,
);

```

image-frame-steps.js - Get image frame IDs.

```

import {
  MedicalImagingClient,
  GetImageSetMetadataCommand,
} from "@aws-sdk/client-medical-imaging";
import { gunzip } from "zlib";
import { promisify } from "util";

import {
  ScenarioAction,
  ScenarioOutput,
} from "@aws-doc-sdk-examples/lib/scenario/index.js";

const gunzipAsync = promisify(gunzip);

/**
 * @typedef {Object} DICOMValueRepresentation
 * @property {string} name
 * @property {string} type
 * @property {string} value
 */

```



```
/**
 * @typedef {Object} ImageFrameInformation
 * @property {string} ID
 * @property {Array<{ Checksum: number, Height: number, Width: number }>}
PixelDataChecksumFromBaseToFullResolution
 * @property {number} MinPixelValue
 * @property {number} MaxPixelValue
 * @property {number} FrameSizeInBytes
 */

/**
 * @typedef {Object} DICOMMetadata
 * @property {Object} DICOM
 * @property {DICOMValueRepresentation[]} DICOMVRs
 * @property {ImageFrameInformation[]} ImageFrames
 */

/**
 * @typedef {Object} Series
 * @property {{ [key: string]: DICOMMetadata }} Instances
 */

/**
 * @typedef {Object} Study
 * @property {Object} DICOM
 * @property {Series[]} Series
 */

/**
 * @typedef {Object} Patient
 * @property {Object} DICOM
 */

/**
 * @typedef {{
 * SchemaVersion: string,
 * DatastoreID: string,
 * ImageSetID: string,
 * Patient: Patient,
 * Study: Study
 * }} ImageSetMetadata
 */
```

```
/**
 * @typedef {{ stackOutputs: {
 *   BucketName: string,
 *   DatastoreID: string,
 *   RoleArn: string
 * }, imageSetIds: string[] }} State
 */

const medicalImagingClient = new MedicalImagingClient({});

export const getImageSetMetadata = new ScenarioAction(
  "getImageSetMetadata",
  async (** @type {State} */ state) => {
    const outputMetadata = [];

    for (const imageSetId of state.imageSetIds) {
      const command = new GetImageSetMetadataCommand({
        datastoreId: state.stackOutputs.DatastoreID,
        imageSetId,
      });

      const response = await medicalImagingClient.send(command);
      const compressedMetadataBlob =
        await response.imageSetMetadataBlob.transformToByteArray();
      const decompressedMetadata = await gunzipAsync(compressedMetadataBlob);
      const imageSetMetadata = JSON.parse(decompressedMetadata.toString());

      outputMetadata.push(imageSetMetadata);
    }

    state.imageSetMetadata = outputMetadata;
  },
);

export const outputImageFrameIds = new ScenarioOutput(
  "outputImageFrameIds",
  (** @type {State & { imageSetMetadata: ImageSetMetadata[] }} */ state) => {
    let output = "";

    for (const metadata of state.imageSetMetadata) {
      const imageSetId = metadata.ImageSetID;
      /** @type {DICOMMetadata[]} */
      const instances = Object.values(metadata.Study.Series).flatMap(
        (series) => {
```

```

        return Object.values(series.Instances);
    },
);
const imageFrameIds = instances.flatMap((instance) =>
    instance.ImageFrames.map((frame) => frame.ID),
);

    output += `Image set ID: ${imageSetId}\nImage frame IDs:\n
${imageFrameIds.join(
    "\n",
    )}\n\n`;
}

    return output;
},
{ slow: false },
);

```

verify-steps.js - Verify image frames. The [AWS HealthImaging Pixel Data Verification](#) library was used for verification.

```

import { spawn } from "node:child_process";

import {
    ScenarioAction,
    ScenarioInput,
} from "@aws-doc-sdk-examples/lib/scenario/index.js";

/**
 * @typedef {Object} DICOMValueRepresentation
 * @property {string} name
 * @property {string} type
 * @property {string} value
 */

/**
 * @typedef {Object} ImageFrameInformation
 * @property {string} ID
 * @property {Array<{ Checksum: number, Height: number, Width: number }>}
PixelDataChecksumFromBaseToFullResolution
 * @property {number} MinPixelValue
 * @property {number} MaxPixelValue

```

```
* @property {number} FrameSizeInBytes
*/

/**
 * @typedef {Object} DICOMMetadata
 * @property {Object} DICOM
 * @property {DICOMValueRepresentation[]} DICOMVRs
 * @property {ImageFrameInformation[]} ImageFrames
 */

/**
 * @typedef {Object} Series
 * @property {{ [key: string]: DICOMMetadata }} Instances
 */

/**
 * @typedef {Object} Study
 * @property {Object} DICOM
 * @property {Series[]} Series
 */

/**
 * @typedef {Object} Patient
 * @property {Object} DICOM
 */

/**
 * @typedef {{
 *   SchemaVersion: string,
 *   DatastoreID: string,
 *   ImageSetID: string,
 *   Patient: Patient,
 *   Study: Study
 * }} ImageSetMetadata
 */

/**
 * @typedef {{ stackOutputs: {
 *   BucketName: string,
 *   DatastoreID: string,
 *   RoleArn: string
 * }, imageSetMetadata: ImageSetMetadata[] }} State
 */
```

```
export const doVerify = new ScenarioInput(
  "doVerify",
  "Do you want to verify the imported images?",
  {
    type: "confirm",
    default: true,
  },
);

export const decodeAndVerifyImages = new ScenarioAction(
  "decodeAndVerifyImages",
  async (** @type {State} */ state) => {
    if (!state.doVerify) {
      process.exit(0);
    }
    const verificationTool = "./pixel-data-verification/index.js";

    for (const metadata of state.imageSetMetadata) {
      const datastoreId = state.stackOutputs.DatastoreId;
      const imageSetId = metadata.ImageSetId;

      for (const [seriesInstanceId, series] of Object.entries(
        metadata.Study.Series,
      )) {
        for (const [sopInstanceId, _] of Object.entries(series.Instances)) {
          console.log(
            `Verifying image set ${imageSetId} with series ${seriesInstanceId} and
sop ${sopInstanceId}`,
          );
          const child = spawn(
            "node",
            [
              verificationTool,
              datastoreId,
              imageSetId,
              seriesInstanceId,
              sopInstanceId,
            ],
            { stdio: "inherit" },
          );
          await new Promise((resolve, reject) => {
            child.on("exit", (code) => {
              if (code === 0) {
```

```

        resolve();
    } else {
        reject(
            new Error(
                `Verification tool exited with code ${code} for image set
                ${imageSetId}`,
            ),
        );
    }
    });
});
}
}
},
);

```

clean-up-steps.js - Destroy resources.

```

import {
    CloudFormationClient,
    DeleteStackCommand,
} from "@aws-sdk/client-cloudformation";
import {
    MedicalImagingClient,
    DeleteImageSetCommand,
} from "@aws-sdk/client-medical-imaging";

import {
    ScenarioAction,
    ScenarioInput,
} from "@aws-doc-sdk-examples/lib/scenario/index.js";

/**
 * @typedef {Object} DICOMValueRepresentation
 * @property {string} name
 * @property {string} type
 * @property {string} value
 */

/**
 * @typedef {Object} ImageFrameInformation

```

```
* @property {string} ID
* @property {Array<{ Checksum: number, Height: number, Width: number }>}
PixelDataChecksumFromBaseToFullResolution
* @property {number} MinPixelValue
* @property {number} MaxPixelValue
* @property {number} FrameSizeInBytes
*/

/**
* @typedef {Object} DICOMMetadata
* @property {Object} DICOM
* @property {DICOMValueRepresentation[]} DICOMVRs
* @property {ImageFrameInformation[]} ImageFrames
*/

/**
* @typedef {Object} Series
* @property {{ [key: string]: DICOMMetadata }} Instances
*/

/**
* @typedef {Object} Study
* @property {Object} DICOM
* @property {Series[]} Series
*/

/**
* @typedef {Object} Patient
* @property {Object} DICOM
*/

/**
* @typedef {{
*   SchemaVersion: string,
*   DatastoreID: string,
*   ImageSetID: string,
*   Patient: Patient,
*   Study: Study
* }} ImageSetMetadata
*/

/**
* @typedef {{ stackOutputs: {
*   BucketName: string,
```

```
*   DatastoreID: string,
*   RoleArn: string
* }, imageSetMetadata: ImageSetMetadata[] ]} State
*/

const cfnClient = new CloudFormationClient({});
const medicalImagingClient = new MedicalImagingClient({});

export const confirmCleanup = new ScenarioInput(
  "confirmCleanup",
  "Do you want to delete the created resources?",
  { type: "confirm" },
);

export const deleteImageSets = new ScenarioAction(
  "deleteImageSets",
  async (/** @type {State} */ state) => {
    const datastoreId = state.stackOutputs.DatastoreID;

    for (const metadata of state.imageSetMetadata) {
      const command = new DeleteImageSetCommand({
        datastoreId,
        imageSetId: metadata.ImageSetID,
      });

      try {
        await medicalImagingClient.send(command);
        console.log(`Successfully deleted image set ${metadata.ImageSetID}`);
      } catch (e) {
        if (e instanceof Error) {
          if (e.name === "ConflictException") {
            console.log(`Image set ${metadata.ImageSetID} already deleted`);
          }
        }
      }
    }
  },
  {
    skipWhen: (/** @type {{{}} */ state) => !state.confirmCleanup,
  },
);

export const deleteStack = new ScenarioAction(
  "deleteStack",
```



```
async (/** @type {State} */ state) => {
  const stackName = state.getStackName;

  const command = new DeleteStackCommand({
    StackName: stackName,
  });

  await cfnClient.send(command);
  console.log(`Stack ${stackName} deletion initiated`);
},
{
  skipWhen: (/** @type {{}} */ state) => !state.confirmCleanup,
},
);
```

- For API details, see the following topics in *AWS SDK for JavaScript API Reference*.
 - [DeleteImageSet](#)
 - [GetDICOMImportJob](#)
 - [GetImageFrame](#)
 - [GetImageSetMetadata](#)
 - [SearchImageSets](#)
 - [StartDICOMImportJob](#)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Tagging a data store

The following code example shows how to tag a HealthImaging data store.

SDK for JavaScript (v3)

To tag a data store.

```
try {
  const datastoreArn =
```

```

    "arn:aws:medical-imaging:us-
east-1:123456789012:datastore/12345678901234567890123456789012";
    const tags = {
      Deployment: "Development",
    };
    await tagResource(datastoreArn, tags);
  } catch (e) {
    console.log(e);
  }
}

```

The utility function for tagging a resource.

```

import { TagResourceCommand } from "@aws-sdk/client-medical-imaging";
import { medicalImagingClient } from "../libs/medicalImagingClient.js";

/**
 * @param {string} resourceArn - The Amazon Resource Name (ARN) for the data store
 * or image set.
 * @param {Record<string,string>} tags - The tags to add to the resource as JSON.
 * - For example: {"Deployment" : "Development"}
 */
export const tagResource = async (
  resourceArn = "arn:aws:medical-imaging:us-east-1:xxxxxx:datastore/xxxxx/imageset/
xxx",
  tags = {}
) => {
  const response = await medicalImagingClient.send(
    new TagResourceCommand({ resourceArn: resourceArn, tags: tags })
  );
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 204,
  //     requestId: '8a6de9a3-ec8e-47ef-8643-473518b19d45',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   }
  // }
  // }

  return response;
}

```

```
};
```

To list tags for a data store.

```
try {
  const datastoreArn =
    "arn:aws:medical-imaging:us-
east-1:123456789012:datastore/12345678901234567890123456789012";
  const { tags } = await listTagsForResource(datastoreArn);
  console.log(tags);
} catch (e) {
  console.log(e);
}
```

The utility function for listing a resource's tags.

```
import { ListTagsForResourceCommand } from "@aws-sdk/client-medical-imaging";
import { medicalImagingClient } from "../libs/medicalImagingClient.js";

/**
 * @param {string} resourceArn - The Amazon Resource Name (ARN) for the data store
 * or image set.
 */
export const listTagsForResource = async (
  resourceArn = "arn:aws:medical-imaging:us-east-1:abc:datastore/def/imageset/ghi"
) => {
  const response = await medicalImagingClient.send(
    new ListTagsForResourceCommand({ resourceArn: resourceArn })
  );
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: '008fc6d3-abec-4870-a155-20fa3631e645',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  //   tags: { Deployment: 'Development' }
  // }
}
```

```
    return response;
  };
```

To untag a data store.

```
try {
  const datastoreArn =
    "arn:aws:medical-imaging:us-
east-1:123456789012:datastore/12345678901234567890123456789012";
  const keys = ["Deployment"];
  await untagResource(datastoreArn, keys);
} catch (e) {
  console.log(e);
}
```

The utility function for untagging a resource.

```
import { UntagResourceCommand } from "@aws-sdk/client-medical-imaging";
import { medicalImagingClient } from "../libs/medicalImagingClient.js";

/**
 * @param {string} resourceArn - The Amazon Resource Name (ARN) for the data store
or image set.
 * @param {string[]} tagKeys - The keys of the tags to remove.
 */
export const untagResource = async (
  resourceArn = "arn:aws:medical-imaging:us-east-1:xxxxxx:datastore/xxxxx/imageset/
xxx",
  tagKeys = []
) => {
  const response = await medicalImagingClient.send(
    new UntagResourceCommand({ resourceArn: resourceArn, tagKeys: tagKeys })
  );
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 204,
  //     requestId: '8a6de9a3-ec8e-47ef-8643-473518b19d45',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
```

```
//      attempts: 1,  
//      totalRetryDelay: 0  
//    }  
//  }  
  
  return response;  
};
```

- For API details, see the following topics in *AWS SDK for JavaScript API Reference*.
 - [ListTagsForResource](#)
 - [TagResource](#)
 - [UntagResource](#)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Tagging an image set

The following code example shows how to tag a HealthImaging image set.

SDK for JavaScript (v3)

To tag an image set.

```
try {  
  const imagesetArn =  
    "arn:aws:medical-imaging:us-  
east-1:123456789012: datastore/12345678901234567890123456789012/  
imageset/12345678901234567890123456789012";  
  const tags = {  
    Deployment: "Development",  
  };  
  await tagResource(imagesetArn, tags);  
} catch (e) {  
  console.log(e);  
}
```

The utility function for tagging a resource.

```
import { TagResourceCommand } from "@aws-sdk/client-medical-imaging";
import { medicalImagingClient } from "../libs/medicalImagingClient.js";

/**
 * @param {string} resourceArn - The Amazon Resource Name (ARN) for the data store
 * or image set.
 * @param {Record<string,string>} tags - The tags to add to the resource as JSON.
 * - For example: {"Deployment" : "Development"}
 */
export const tagResource = async (
  resourceArn = "arn:aws:medical-imaging:us-east-1:xxxxxx:datastore/xxxxx/imageset/
  xxx",
  tags = {}
) => {
  const response = await medicalImagingClient.send(
    new TagResourceCommand({ resourceArn: resourceArn, tags: tags })
  );
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 204,
  //     requestId: '8a6de9a3-ec8e-47ef-8643-473518b19d45',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   }
  // }

  return response;
};
```

To list tags for an image set.

```
try {
  const imagesetArn =
    "arn:aws:medical-imaging:us-
    east-1:123456789012:datastore/12345678901234567890123456789012/
    imageset/12345678901234567890123456789012";
  const { tags } = await listTagsForResource(imagesetArn);
```

```
    console.log(tags);
  } catch (e) {
    console.log(e);
  }
}
```

The utility function for listing a resource's tags.

```
import { ListTagsForResourceCommand } from "@aws-sdk/client-medical-imaging";
import { medicalImagingClient } from "../libs/medicalImagingClient.js";

/**
 * @param {string} resourceArn - The Amazon Resource Name (ARN) for the data store
 * or image set.
 */
export const listTagsForResource = async (
  resourceArn = "arn:aws:medical-imaging:us-east-1:abc:datastore/def/imageset/ghi"
) => {
  const response = await medicalImagingClient.send(
    new ListTagsForResourceCommand({ resourceArn: resourceArn })
  );
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: '008fc6d3-abec-4870-a155-20fa3631e645',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  //   tags: { Deployment: 'Development' }
  // }

  return response;
};
```

To untag an image set.

```
try {
  const imagesetArn =
```

```

    "arn:aws:medical-imaging:us-
east-1:123456789012:datastore/12345678901234567890123456789012/
imageset/12345678901234567890123456789012";
    const keys = ["Deployment"];
    await untagResource(imagesetArn, keys);
  } catch (e) {
    console.log(e);
  }
}

```

The utility function for untagging a resource.

```

import { UntagResourceCommand } from "@aws-sdk/client-medical-imaging";
import { medicalImagingClient } from "../libs/medicalImagingClient.js";

/**
 * @param {string} resourceArn - The Amazon Resource Name (ARN) for the data store
or image set.
 * @param {string[]} tagKeys - The keys of the tags to remove.
 */
export const untagResource = async (
  resourceArn = "arn:aws:medical-imaging:us-east-1:xxxxxx:datastore/xxxxx/imageset/
xxx",
  tagKeys = []
) => {
  const response = await medicalImagingClient.send(
    new UntagResourceCommand({ resourceArn: resourceArn, tagKeys: tagKeys })
  );
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 204,
  //     requestId: '8a6de9a3-ec8e-47ef-8643-473518b19d45',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   }
  // }
  // }

  return response;
};

```


- For API details, see the following topics in *AWS SDK for JavaScript API Reference*.
 - [ListTagsForResource](#)
 - [TagResource](#)
 - [UntagResource](#)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

IAM examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with IAM.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Scenarios are code examples that show you how to accomplish specific tasks by calling multiple functions within a service or combined with other AWS services.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Get started

Hello IAM

The following code examples show how to get started using IAM.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { IAMClient, paginateListPolicies } from "@aws-sdk/client-iam";
```

```
const client = new IAMClient({});

export const listLocalPolicies = async () => {
  /**
   * In v3, the clients expose paginateOperationName APIs that are written using
   * async generators so that you can use async iterators in a for await..of loop.
   * https://docs.aws.amazon.com/AWSJavaScriptSDK/v3/latest/index.html#paginators
   */
  const paginator = paginateListPolicies(
    { client, pageSize: 10 },
    // List only customer managed policies.
    { Scope: "Local" },
  );

  console.log("IAM policies defined in your account:");
  let policyCount = 0;
  for await (const page of paginator) {
    if (page.Policies) {
      page.Policies.forEach((p) => {
        console.log(`${p.PolicyName}`);
        policyCount++;
      });
    }
  }
  console.log(`Found ${policyCount} policies.`);
};
```

- For API details, see [ListPolicies](#) in *AWS SDK for JavaScript API Reference*.

Topics

- [Actions](#)
- [Scenarios](#)

Actions

AttachRolePolicy

The following code example shows how to use `AttachRolePolicy`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Attach the policy.

```
import { AttachRolePolicyCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
 * @param {string} policyArn
 * @param {string} roleName
 */
export const attachRolePolicy = (policyArn, roleName) => {
  const command = new AttachRolePolicyCommand({
    PolicyArn: policyArn,
    RoleName: roleName,
  });

  return client.send(command);
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [AttachRolePolicy](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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");
      }
    });
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [AttachRolePolicy](#) in *AWS SDK for JavaScript API Reference*.

CreateAccessKey

The following code example shows how to use CreateAccessKey.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create the access key.

```
import { CreateAccessKeyCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
 * @param {string} userName
 */
export const createAccessKey = (userName) => {
  const command = new CreateAccessKeyCommand({ Username: userName });
  return client.send(command);
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [CreateAccessKey](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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);
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [CreateAccessKey](#) in *AWS SDK for JavaScript API Reference*.

CreateAccountAlias

The following code example shows how to use CreateAccountAlias.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create the account alias.

```
import { CreateAccountAliasCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
 * @param {string} alias - A unique name for the account alias.
 * @returns
```

```
*/
export const createAccountAlias = (alias) => {
  const command = new CreateAccountAliasCommand({
    AccountAlias: alias,
  });

  return client.send(command);
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [CreateAccountAlias](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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);
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [CreateAccountAlias](#) in *AWS SDK for JavaScript API Reference*.

CreateGroup

The following code example shows how to use CreateGroup.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { CreateGroupCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
 * @param {string} groupName
 */
export const createGroup = async (groupName) => {
  const command = new CreateGroupCommand({ GroupName: groupName });

  const response = await client.send(command);
  console.log(response);
  return response;
};
```

- For API details, see [CreateGroup](#) in *AWS SDK for JavaScript API Reference*.

CreateInstanceProfile

The following code example shows how to use CreateInstanceProfile.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const { InstanceProfile } = await iamClient.send(
  new CreateInstanceProfileCommand({
    InstanceProfileName: NAMES.ssmOnlyInstanceProfileName,
  }),
);
await waitUntilInstanceProfileExists(
  { client: iamClient },
  { InstanceProfileName: NAMES.ssmOnlyInstanceProfileName },
);
```

- For API details, see [CreateInstanceProfile](#) in *AWS SDK for JavaScript API Reference*.

CreatePolicy

The following code example shows how to use CreatePolicy.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create the policy.

```
import { CreatePolicyCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
```

```
* @param {string} policyName
*/
export const createPolicy = (policyName) => {
  const command = new CreatePolicyCommand({
    PolicyDocument: JSON.stringify({
      Version: "2012-10-17",
      Statement: [
        {
          Effect: "Allow",
          Action: "*",
          Resource: "*",
        },
      ],
    }),
    PolicyName: policyName,
  });

  return client.send(command);
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [CreatePolicy](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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);
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [CreatePolicy](#) in *AWS SDK for JavaScript API Reference*.

CreateRole

The following code example shows how to use `CreateRole`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create the role.

```
import { CreateRoleCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
 * @param {string} roleName
 */
export const createRole = (roleName) => {
  const command = new CreateRoleCommand({
    AssumeRolePolicyDocument: JSON.stringify({
      Version: "2012-10-17",
      Statement: [
        {
          Effect: "Allow",
          Principal: {
            Service: "lambda.amazonaws.com",
          },
          Action: "sts:AssumeRole",
        },
      ],
    }),
    RoleName: roleName,
  });

  return client.send(command);
};
```

- For API details, see [CreateRole](#) in *AWS SDK for JavaScript API Reference*.

CreateSAMLProvider

The following code example shows how to use CreateSAMLProvider.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { CreateSAMLProviderCommand, IAMClient } from "@aws-sdk/client-iam";
import { readFileSync } from "fs";
import * as path from "path";
import { dirnameFromMetaUrl } from "@aws-doc-sdk-examples/lib/utils/util-fs.js";

const client = new IAMClient({});

/**
 * This sample document was generated using Auth0.
 * For more information on generating this document,
 * see https://docs.aws.amazon.com/IAM/latest/UserGuide/
 * id_roles_providers_create_saml.html#samlstep1.
 */
const sampleMetadataDocument = readFileSync(
  path.join(
    dirnameFromMetaUrl(import.meta.url),
    "../../../../../resources/sample_files/sample_saml_metadata.xml",
  ),
);

/**
 *
 * @param {*} providerName
 * @returns
 */
export const createSAMLProvider = async (providerName) => {
  const command = new CreateSAMLProviderCommand({
    Name: providerName,
    SAMLMetadataDocument: sampleMetadataDocument.toString(),
  });
```

```
const response = await client.send(command);
console.log(response);
return response;
};
```

- For API details, see [CreateSAMLProvider](#) in *AWS SDK for JavaScript API Reference*.

CreateServiceLinkedRole

The following code example shows how to use `CreateServiceLinkedRole`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create a service-linked role.

```
import {
  CreateServiceLinkedRoleCommand,
  GetRoleCommand,
  IAMClient,
} from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
 * @param {string} serviceName
 */
export const createServiceLinkedRole = async (serviceName) => {
  const command = new CreateServiceLinkedRoleCommand({
    // For a list of AWS services that support service-linked roles,
    // see https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_aws-services-
    // that-work-with-iam.html.
    //
    // For a list of AWS service endpoints, see https://docs.aws.amazon.com/general/
    // latest/gr/aws-service-information.html.
  });
```

```
    AWSServiceName: serviceName,
  });
  try {
    const response = await client.send(command);
    console.log(response);
    return response;
  } catch (caught) {
    if (
      caught instanceof Error &&
      caught.name === "InvalidInputException" &&
      caught.message.includes(
        "Service role name AWSServiceRoleForElasticBeanstalk has been taken in this
account",
      )
    ) {
      console.warn(caught.message);
      return client.send(
        new GetRoleCommand({ RoleName: "AWSServiceRoleForElasticBeanstalk" }),
      );
    } else {
      throw caught;
    }
  }
};
```

- For API details, see [CreateServiceLinkedRole](#) in *AWS SDK for JavaScript API Reference*.

CreateUser

The following code example shows how to use CreateUser.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create the user.

```
import { CreateUserCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
 * @param {string} name
 */
export const createUser = (name) => {
  const command = new CreateUserCommand({ UserName: name });
  return client.send(command);
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [CreateUser](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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
  );
}
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [CreateUser](#) in *AWS SDK for JavaScript API Reference*.

DeleteAccessKey

The following code example shows how to use DeleteAccessKey.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Delete the access key.

```
import { DeleteAccessKeyCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
 * @param {string} userName
 * @param {string} accessKeyId
 */
export const deleteAccessKey = (userName, accessKeyId) => {
  const command = new DeleteAccessKeyCommand({
    AccessKeyId: accessKeyId,
```

```
    UserName: userName,
  });

  return client.send(command);
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [DeleteAccessKey](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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);
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [DeleteAccessKey](#) in *AWS SDK for JavaScript API Reference*.

DeleteAccountAlias

The following code example shows how to use DeleteAccountAlias.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Delete the account alias.

```
import { DeleteAccountAliasCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
 * @param {string} alias
 */
export const deleteAccountAlias = (alias) => {
  const command = new DeleteAccountAliasCommand({ AccountAlias: alias });

  return client.send(command);
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [DeleteAccountAlias](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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);
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [DeleteAccountAlias](#) in *AWS SDK for JavaScript API Reference*.

DeleteGroup

The following code example shows how to use DeleteGroup.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { DeleteGroupCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
 * @param {string} groupName
 */
export const deleteGroup = async (groupName) => {
  const command = new DeleteGroupCommand({
```

```
    GroupName: groupName,
  });

  const response = await client.send(command);
  console.log(response);
  return response;
};
```

- For API details, see [DeleteGroup](#) in *AWS SDK for JavaScript API Reference*.

DeleteInstanceProfile

The following code example shows how to use DeleteInstanceProfile.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const client = new IAMClient({});
await client.send(
  new DeleteInstanceProfileCommand({
    InstanceProfileName: NAMES.instanceProfileName,
  }),
);
```

- For API details, see [DeleteInstanceProfile](#) in *AWS SDK for JavaScript API Reference*.

DeletePolicy

The following code example shows how to use DeletePolicy.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Delete the policy.

```
import { DeletePolicyCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
 * @param {string} policyArn
 */
export const deletePolicy = (policyArn) => {
  const command = new DeletePolicyCommand({ PolicyArn: policyArn });
  return client.send(command);
};
```

- For API details, see [DeletePolicy](#) in *AWS SDK for JavaScript API Reference*.

DeleteRole

The following code example shows how to use DeleteRole.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Delete the role.

```
import { DeleteRoleCommand, IAMClient } from "@aws-sdk/client-iam";
```

```
const client = new IAMClient({});

/**
 *
 * @param {string} roleName
 */
export const deleteRole = (roleName) => {
  const command = new DeleteRoleCommand({ RoleName: roleName });
  return client.send(command);
};
```

- For API details, see [DeleteRole](#) in *AWS SDK for JavaScript API Reference*.

DeleteRolePolicy

The following code example shows how to use DeleteRolePolicy.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { DeleteRolePolicyCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
 * @param {string} roleName
 * @param {string} policyName
 */
export const deleteRolePolicy = (roleName, policyName) => {
  const command = new DeleteRolePolicyCommand({
    RoleName: roleName,
    PolicyName: policyName,
  });
};
```

```
    return client.send(command);
  };
```

- For API details, see [DeleteRolePolicy](#) in *AWS SDK for JavaScript API Reference*.

DeleteSAMLProvider

The following code example shows how to use DeleteSAMLProvider.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { DeleteSAMLProviderCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
 * @param {string} providerArn
 * @returns
 */
export const deleteSAMLProvider = async (providerArn) => {
  const command = new DeleteSAMLProviderCommand({
    SAMLProviderArn: providerArn,
  });

  const response = await client.send(command);
  console.log(response);
  return response;
};
```

- For API details, see [DeleteSAMLProvider](#) in *AWS SDK for JavaScript API Reference*.

DeleteServerCertificate

The following code example shows how to use DeleteServerCertificate.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Delete a server certificate.

```
import { DeleteServerCertificateCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
 * @param {string} certName
 */
export const deleteServerCertificate = (certName) => {
  const command = new DeleteServerCertificateCommand({
    ServerCertificateName: certName,
  });

  return client.send(command);
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [DeleteServerCertificate](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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);
    }
  }
);
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [DeleteServerCertificate](#) in *AWS SDK for JavaScript API Reference*.

DeleteServiceLinkedRole

The following code example shows how to use DeleteServiceLinkedRole.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { DeleteServiceLinkedRoleCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
```

```
*
* @param {string} roleName
*/
export const deleteServiceLinkedRole = (roleName) => {
  const command = new DeleteServiceLinkedRoleCommand({ RoleName: roleName });
  return client.send(command);
};
```

- For API details, see [DeleteServiceLinkedRole](#) in *AWS SDK for JavaScript API Reference*.

DeleteUser

The following code example shows how to use DeleteUser.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Delete the user.

```
import { DeleteUserCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
 * @param {string} name
 */
export const deleteUser = (name) => {
  const command = new DeleteUserCommand({ UserName: name });
  return client.send(command);
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [DeleteUser](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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);
      }
    });
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [DeleteUser](#) in *AWS SDK for JavaScript API Reference*.

DetachRolePolicy

The following code example shows how to use `DetachRolePolicy`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Detach the policy.

```
import { DetachRolePolicyCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
 * @param {string} policyArn
 * @param {string} roleName
 */
export const detachRolePolicy = (policyArn, roleName) => {
  const command = new DetachRolePolicyCommand({
    PolicyArn: policyArn,
    RoleName: roleName,
  });

  return client.send(command);
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [DetachRolePolicy](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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();
          }
        });
      }
    });
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [DetachRolePolicy](#) in *AWS SDK for JavaScript API Reference*.

GetAccessKeyLastUsed

The following code example shows how to use `GetAccessKeyLastUsed`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Get the access key.

```
import { GetAccessKeyLastUsedCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
 * @param {string} accessKeyId
 */
export const getAccessKeyLastUsed = async (accessKeyId) => {
  const command = new GetAccessKeyLastUsedCommand({
    AccessKeyId: accessKeyId,
  });

  const response = await client.send(command);

  if (response.AccessKeyLastUsed?.LastUsedDate) {
    console.log(`
    ${accessKeyId} was last used by ${response.UserName} via
    the ${response.AccessKeyLastUsed.ServiceName} service on
    ${response.AccessKeyLastUsed.LastUsedDate.toISOString()}
    `);
  }

  return response;
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [GetAccessKeyLastUsed](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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);
    }
  }
);
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [GetAccessKeyLastUsed](#) in *AWS SDK for JavaScript API Reference*.

GetAccountPasswordPolicy

The following code example shows how to use `GetAccountPasswordPolicy`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Get the account password policy.

```
import {
  GetAccountPasswordPolicyCommand,
  IAMClient,
} from "@aws-sdk/client-iam";

const client = new IAMClient({});

export const getAccountPasswordPolicy = async () => {
  const command = new GetAccountPasswordPolicyCommand({});

  const response = await client.send(command);
  console.log(response.PasswordPolicy);
  return response;
};
```

- For API details, see [GetAccountPasswordPolicy](#) in *AWS SDK for JavaScript API Reference*.

GetPolicy

The following code example shows how to use `GetPolicy`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Get the policy.

```
import { GetPolicyCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
 * @param {string} policyArn
 */
export const getPolicy = (policyArn) => {
  const command = new GetPolicyCommand({
    PolicyArn: policyArn,
  });

  return client.send(command);
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [GetPolicy](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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);
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [GetPolicy](#) in *AWS SDK for JavaScript API Reference*.

GetRole

The following code example shows how to use `GetRole`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Get the role.

```
import { GetRoleCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
 * @param {string} roleName
 */
export const getRole = (roleName) => {
  const command = new GetRoleCommand({
    RoleName: roleName,
  });

  return client.send(command);
};
```

- For API details, see [GetRole](#) in *AWS SDK for JavaScript API Reference*.

GetServerCertificate

The following code example shows how to use `GetServerCertificate`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Get a server certificate.

```
import { GetServerCertificateCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
 * @param {string} certName
 * @returns
 */
export const getServerCertificate = async (certName) => {
  const command = new GetServerCertificateCommand({
    ServerCertificateName: certName,
  });

  const response = await client.send(command);
  console.log(response);
  return response;
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [GetServerCertificate](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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);
    }
  }
);
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [GetServerCertificate](#) in *AWS SDK for JavaScript API Reference*.

GetServiceLinkedRoleDeletionStatus

The following code example shows how to use `GetServiceLinkedRoleDeletionStatus`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import {
  GetServiceLinkedRoleDeletionStatusCommand,
  IAMClient,
} from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
 * @param {string} deletionTaskId
 */
export const getServiceLinkedRoleDeletionStatus = (deletionTaskId) => {
  const command = new GetServiceLinkedRoleDeletionStatusCommand({
    DeletionTaskId: deletionTaskId,
  });

  return client.send(command);
};
```

- For API details, see [GetServiceLinkedRoleDeletionStatus](#) in *AWS SDK for JavaScript API Reference*.

ListAccessKeys

The following code example shows how to use `ListAccessKeys`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

List the access keys.

```
import { ListAccessKeysCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 * A generator function that handles paginated results.
 * The AWS SDK for JavaScript (v3) provides {@link https://docs.aws.amazon.com/
 * AWSJavaScriptSDK/v3/latest/index.html#paginators | paginator} functions to simplify
 * this.
 *
 * @param {string} userName
 */
export async function* listAccessKeys(userName) {
  const command = new ListAccessKeysCommand({
    MaxItems: 5,
    UserName: userName,
  });

  /**
   * @type {import("@aws-sdk/client-iam").ListAccessKeysCommandOutput | undefined}
   */
  let response = await client.send(command);

  while (response?.AccessKeyMetadata?.length) {
    for (const key of response.AccessKeyMetadata) {
      yield key;
    }

    if (response.IsTruncated) {
      response = await client.send(
        new ListAccessKeysCommand({
          Marker: response.Marker,
        }),
      );
    }
  }
}
```

```
    );  
  } else {  
    break;  
  }  
}  
}
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [ListAccessKeys](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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);  
  }  
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).

- For API details, see [ListAccessKeys](#) in *AWS SDK for JavaScript API Reference*.

ListAccountAliases

The following code example shows how to use ListAccountAliases.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

List the account aliases.

```
import { ListAccountAliasesCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 * A generator function that handles paginated results.
 * The AWS SDK for JavaScript (v3) provides {@link https://docs.aws.amazon.com/AWSJavaScriptSDK/v3/latest/index.html#paginators | paginator} functions to simplify this.
 */
export async function* listAccountAliases() {
  const command = new ListAccountAliasesCommand({ MaxItems: 5 });

  let response = await client.send(command);

  while (response.AccountAliases?.length) {
    for (const alias of response.AccountAliases) {
      yield alias;
    }

    if (response.IsTruncated) {
      response = await client.send(
        new ListAccountAliasesCommand({
          Marker: response.Marker,
          MaxItems: 5,
        }),
      );
    }
  }
}
```

```
    );  
  } else {  
    break;  
  }  
}  
}
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [ListAccountAliases](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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);  
  }  
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [ListAccountAliases](#) in *AWS SDK for JavaScript API Reference*.

ListAttachedRolePolicies

The following code example shows how to use ListAttachedRolePolicies.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

List the policies that are attached to a role.

```
import {
  ListAttachedRolePoliciesCommand,
  IAMClient,
} from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 * A generator function that handles paginated results.
 * The AWS SDK for JavaScript (v3) provides {@link https://docs.aws.amazon.com/
AWSJavaScriptSDK/v3/latest/index.html#paginators | paginator} functions to simplify
this.
 * @param {string} roleName
 */
export async function* listAttachedRolePolicies(roleName) {
  const command = new ListAttachedRolePoliciesCommand({
    RoleName: roleName,
  });

  let response = await client.send(command);

  while (response.AttachedPolicies?.length) {
    for (const policy of response.AttachedPolicies) {
      yield policy;
    }

    if (response.IsTruncated) {
      response = await client.send(
        new ListAttachedRolePoliciesCommand({
```

```
        RoleName: roleName,  
        Marker: response.Marker,  
    }},  
    );  
} else {  
    break;  
}  
}  
}
```

- For API details, see [ListAttachedRolePolicies](#) in *AWS SDK for JavaScript API Reference*.

ListGroups

The following code example shows how to use `ListGroups`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

List the groups.

```
import { ListGroupsCommand, IAMClient } from "@aws-sdk/client-iam";  
  
const client = new IAMClient({});  
  
/**  
 * A generator function that handles paginated results.  
 * The AWS SDK for JavaScript (v3) provides {@link https://docs.aws.amazon.com/  
AWSJavaScriptSDK/v3/latest/index.html#paginators | paginator} functions to simplify  
 this.  
 */  
export async function* listGroups() {  
    const command = new ListGroupsCommand({  
        MaxItems: 10,  
    });
```

```
let response = await client.send(command);

while (response.Groups?.length) {
  for (const group of response.Groups) {
    yield group;
  }

  if (response.IsTruncated) {
    response = await client.send(
      new ListGroupsCommand({
        Marker: response.Marker,
        MaxItems: 10,
      }),
    );
  } else {
    break;
  }
}
```

- For API details, see [ListGroups](#) in *AWS SDK for JavaScript API Reference*.

ListPolicies

The following code example shows how to use `ListPolicies`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

List the policies.

```
import { ListPoliciesCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});
```

```
/**
 * A generator function that handles paginated results.
 * The AWS SDK for JavaScript (v3) provides {@link https://docs.aws.amazon.com/
AWSJavaScriptSDK/v3/latest/index.html#paginator | paginator} functions to simplify
 this.
 *
 */
export async function* listPolicies() {
  const command = new ListPoliciesCommand({
    MaxItems: 10,
    OnlyAttached: false,
    // List only the customer managed policies in your Amazon Web Services account.
    Scope: "Local",
  });

  let response = await client.send(command);

  while (response.Policies?.length) {
    for (const policy of response.Policies) {
      yield policy;
    }

    if (response.IsTruncated) {
      response = await client.send(
        new ListPoliciesCommand({
          Marker: response.Marker,
          MaxItems: 10,
          OnlyAttached: false,
          Scope: "Local",
        })),
    );
  } else {
    break;
  }
}
}
```

- For API details, see [ListPolicies](#) in *AWS SDK for JavaScript API Reference*.

ListRolePolicies

The following code example shows how to use `ListRolePolicies`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

List the policies.

```
import { ListRolePoliciesCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 * A generator function that handles paginated results.
 * The AWS SDK for JavaScript (v3) provides {@link https://docs.aws.amazon.com/
 * AWSJavaScriptSDK/v3/latest/index.html#paginators | paginator} functions to simplify
 * this.
 *
 * @param {string} roleName
 */
export async function* listRolePolicies(roleName) {
  const command = new ListRolePoliciesCommand({
    RoleName: roleName,
    MaxItems: 10,
  });

  let response = await client.send(command);

  while (response.PolicyNames?.length) {
    for (const policyName of response.PolicyNames) {
      yield policyName;
    }

    if (response.IsTruncated) {
      response = await client.send(
        new ListRolePoliciesCommand({
          RoleName: roleName,
          MaxItems: 10,
          Marker: response.Marker,
        })
      );
    }
  }
}
```

```
    } else {  
      break;  
    }  
  }  
}
```

- For API details, see [ListRolePolicies](#) in *AWS SDK for JavaScript API Reference*.

ListRoles

The following code example shows how to use `ListRoles`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

List the roles.

```
import { ListRolesCommand, IAMClient } from "@aws-sdk/client-iam";  
  
const client = new IAMClient({});  
  
/**  
 * A generator function that handles paginated results.  
 * The AWS SDK for JavaScript (v3) provides {@link https://docs.aws.amazon.com/  
AWSJavaScriptSDK/v3/latest/index.html#paginators | paginator} functions to simplify  
 this.  
 *  
 */  
export async function* listRoles() {  
  const command = new ListRolesCommand({  
    MaxItems: 10,  
  });  
  
  /**  
   * @type {import("@aws-sdk/client-iam").ListRolesCommandOutput | undefined}  
   */
```



```
let response = await client.send(command);

while (response?.Roles?.length) {
  for (const role of response.Roles) {
    yield role;
  }

  if (response.IsTruncated) {
    response = await client.send(
      new ListRolesCommand({
        Marker: response.Marker,
      }),
    );
  } else {
    break;
  }
}
}
```

- For API details, see [ListRoles](#) in *AWS SDK for JavaScript API Reference*.

ListSAMLProviders

The following code example shows how to use `ListSAMLProviders`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

List the SAML providers.

```
import { ListSAMLProvidersCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

export const listSamlProviders = async () => {
```

```
const command = new ListSAMLProvidersCommand({});

const response = await client.send(command);
console.log(response);
return response;
};
```

- For API details, see [ListSAMLProviders](#) in *AWS SDK for JavaScript API Reference*.

ListServerCertificates

The following code example shows how to use `ListServerCertificates`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

List the certificates.

```
import { ListServerCertificatesCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 * A generator function that handles paginated results.
 * The AWS SDK for JavaScript (v3) provides {@link https://docs.aws.amazon.com/AWSJavaScriptSDK/v3/latest/index.html#paginators | paginator} functions to simplify this.
 *
 */
export async function* listServerCertificates() {
  const command = new ListServerCertificatesCommand({});
  let response = await client.send(command);

  while (response.ServerCertificateMetadataList?.length) {
    for await (const cert of response.ServerCertificateMetadataList) {
      yield cert;
    }
  }
}
```

```
    }

    if (response.IsTruncated) {
        response = await client.send(new ListServerCertificatesCommand({}));
    } else {
        break;
    }
}
}
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [ListServerCertificates](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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);
    }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [ListServerCertificates](#) in *AWS SDK for JavaScript API Reference*.

ListUsers

The following code example shows how to use ListUsers.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

List the users.

```
import { ListUsersCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

export const listUsers = async () => {
  const command = new ListUsersCommand({ MaxItems: 10 });

  const response = await client.send(command);
  response.Users?.forEach(({ UserName, CreateDate }) => {
    console.log(`${UserName} created on: ${CreateDate}`);
  });
  return response;
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [ListUsers](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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);
    });
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [ListUsers](#) in *AWS SDK for JavaScript API Reference*.

PutRolePolicy

The following code example shows how to use PutRolePolicy.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { PutRolePolicyCommand, IAMClient } from "@aws-sdk/client-iam";

const examplePolicyDocument = JSON.stringify({
```

```
Version: "2012-10-17",
Statement: [
  {
    Sid: "VisualEditor0",
    Effect: "Allow",
    Action: [
      "s3:ListBucketMultipartUploads",
      "s3:ListBucketVersions",
      "s3:ListBucket",
      "s3:ListMultipartUploadParts",
    ],
    Resource: "arn:aws:s3:::some-test-bucket",
  },
  {
    Sid: "VisualEditor1",
    Effect: "Allow",
    Action: [
      "s3:ListStorageLensConfigurations",
      "s3:ListAccessPointsForObjectLambda",
      "s3:ListAllMyBuckets",
      "s3:ListAccessPoints",
      "s3:ListJobs",
      "s3:ListMultiRegionAccessPoints",
    ],
    Resource: "*",
  },
],
});

const client = new IAMClient({});

/**
 *
 * @param {string} roleName
 * @param {string} policyName
 * @param {string} policyDocument
 */
export const putRolePolicy = async (roleName, policyName, policyDocument) => {
  const command = new PutRolePolicyCommand({
    RoleName: roleName,
    PolicyName: policyName,
    PolicyDocument: policyDocument,
  });
```

```
const response = await client.send(command);
console.log(response);
return response;
};
```

- For API details, see [PutRolePolicy](#) in *AWS SDK for JavaScript API Reference*.

UpdateAccessKey

The following code example shows how to use UpdateAccessKey.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Update the access key.

```
import {
  UpdateAccessKeyCommand,
  IAMClient,
  StatusType,
} from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
 * @param {string} userName
 * @param {string} accessKeyId
 */
export const updateAccessKey = (userName, accessKeyId) => {
  const command = new UpdateAccessKeyCommand({
    AccessKeyId: accessKeyId,
    Status: StatusType.Inactive,
    UserName: userName,
  });
```

```
    return client.send(command);
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [UpdateAccessKey](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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);
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [UpdateAccessKey](#) in *AWS SDK for JavaScript API Reference*.

UpdateServerCertificate

The following code example shows how to use UpdateServerCertificate.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Update a server certificate.

```
import { UpdateServerCertificateCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
 * @param {string} currentName
 * @param {string} newName
 */
export const updateServerCertificate = (currentName, newName) => {
  const command = new UpdateServerCertificateCommand({
    ServerCertificateName: currentName,
    NewServerCertificateName: newName,
  });

  return client.send(command);
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [UpdateServerCertificate](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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);
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [UpdateServerCertificate](#) in *AWS SDK for JavaScript API Reference*.

UpdateUser

The following code example shows how to use UpdateUser.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Update the user.

```
import { UpdateUserCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
 * @param {string} currentUserName
 * @param {string} newUserName
 */
export const updateUser = (currentUserName, newUserName) => {
  const command = new UpdateUserCommand({
    UserName: currentUserName,
    NewUserName: newUserName,
  });

  return client.send(command);
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [UpdateUser](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// 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);
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [UpdateUser](#) in *AWS SDK for JavaScript API Reference*.

UploadServerCertificate

The following code example shows how to use UploadServerCertificate.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { UploadServerCertificateCommand, IAMClient } from "@aws-sdk/client-iam";
import { readFileSync } from "fs";
import { dirnameFromMetaUrl } from "@aws-doc-sdk-examples/lib/utils/util-fs.js";
import * as path from "path";
```

```
const client = new IAMClient({});

const certMessage = `Generate a certificate and key with the following command, or
the equivalent for your system.

openssl req -x509 -newkey rsa:4096 -sha256 -days 3650 -nodes \
-keyout example.key -out example.crt -subj "/CN=example.com" \
-addext "subjectAltName=DNS:example.com,DNS:www.example.net,IP:10.0.0.1"
`;

const getCertAndKey = () => {
  try {
    const cert = readFileSync(
      path.join(dirnameFromMetaUrl(import.meta.url), "./example.crt"),
    );
    const key = readFileSync(
      path.join(dirnameFromMetaUrl(import.meta.url), "./example.key"),
    );
    return { cert, key };
  } catch (err) {
    if (err.code === "ENOENT") {
      throw new Error(
        `Certificate and/or private key not found. ${certMessage}`,
      );
    }
  }

  throw err;
};

/**
 *
 * @param {string} certificateName
 */
export const uploadServerCertificate = (certificateName) => {
  const { cert, key } = getCertAndKey();
  const command = new UploadServerCertificateCommand({
    ServerCertificateName: certificateName,
    CertificateBody: cert.toString(),
    PrivateKey: key.toString(),
  });

  return client.send(command);
};
```

- For API details, see [UploadServerCertificate](#) in *AWS SDK for JavaScript API Reference*.

Scenarios

Build and manage a resilient service

The following code example shows how to create a load-balanced web service that returns book, movie, and song recommendations. The example shows how the service responds to failures, and how to restructure the service for more resilience when failures occur.

- Use an Amazon EC2 Auto Scaling group to create Amazon Elastic Compute Cloud (Amazon EC2) instances based on a launch template and to keep the number of instances in a specified range.
- Handle and distribute HTTP requests with Elastic Load Balancing.
- Monitor the health of instances in an Auto Scaling group and forward requests only to healthy instances.
- Run a Python web server on each EC2 instance to handle HTTP requests. The web server responds with recommendations and health checks.
- Simulate a recommendation service with an Amazon DynamoDB table.
- Control web server response to requests and health checks by updating AWS Systems Manager parameters.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Run the interactive scenario at a command prompt.

```
#!/usr/bin/env node
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
```

```
import {
  Scenario,
  parseScenarioArgs,
} from "@aws-doc-sdk-examples/lib/scenario/index.js";

/**
 * The workflow steps are split into three stages:
 * - deploy
 * - demo
 * - destroy
 *
 * Each of these stages has a corresponding file prefixed with steps-*.
 */
import { deploySteps } from "./steps-deploy.js";
import { demoSteps } from "./steps-demo.js";
import { destroySteps } from "./steps-destroy.js";

/**
 * The context is passed to every scenario. Scenario steps
 * will modify the context.
 */
const context = {};

/**
 * Three Scenarios are created for the workflow. A Scenario is an orchestration
 class
 * that simplifies running a series of steps.
 */
export const scenarios = {
  // Deploys all resources necessary for the workflow.
  deploy: new Scenario("Resilient Workflow - Deploy", deploySteps, context),
  // Demonstrates how a fragile web service can be made more resilient.
  demo: new Scenario("Resilient Workflow - Demo", demoSteps, context),
  // Destroys the resources created for the workflow.
  destroy: new Scenario("Resilient Workflow - Destroy", destroySteps, context),
};

// Call function if run directly
import { fileURLToPath } from "url";

if (process.argv[1] === fileURLToPath(import.meta.url)) {
  parseScenarioArgs(scenarios);
}
```

Create steps to deploy all of the resources.

```
import { join } from "node:path";
import { readFileSync, writeFileSync } from "node:fs";
import axios from "axios";

import {
  BatchWriteItemCommand,
  CreateTableCommand,
  DynamoDBClient,
  waitUntilTableExists,
} from "@aws-sdk/client-dynamodb";
import {
  EC2Client,
  CreateKeyPairCommand,
  CreateLaunchTemplateCommand,
  DescribeAvailabilityZonesCommand,
  DescribeVpcsCommand,
  DescribeSubnetsCommand,
  DescribeSecurityGroupsCommand,
  AuthorizeSecurityGroupIngressCommand,
} from "@aws-sdk/client-ec2";
import {
  IAMClient,
  CreatePolicyCommand,
  CreateRoleCommand,
  CreateInstanceProfileCommand,
  AddRoleToInstanceProfileCommand,
  AttachRolePolicyCommand,
  waitUntilInstanceProfileExists,
} from "@aws-sdk/client-iam";
import { SSMClient, GetParameterCommand } from "@aws-sdk/client-ssm";
import {
  CreateAutoScalingGroupCommand,
  AutoScalingClient,
  AttachLoadBalancerTargetGroupsCommand,
} from "@aws-sdk/client-auto-scaling";
import {
  CreateListenerCommand,
  CreateLoadBalancerCommand,
  CreateTargetGroupCommand,
```



```
ElasticLoadBalancingV2Client,
waitUntilLoadBalancerAvailable,
} from "@aws-sdk/client-elastic-load-balancing-v2";

import {
  ScenarioOutput,
  ScenarioInput,
  ScenarioAction,
} from "@aws-doc-sdk-examples/lib/scenario/index.js";
import { saveState } from "@aws-doc-sdk-examples/lib/scenario/steps-common.js";
import { retry } from "@aws-doc-sdk-examples/lib/utils/util-timers.js";

import { MESSAGES, NAMES, RESOURCES_PATH, ROOT } from "./constants.js";
import { initParamsSteps } from "./steps-reset-params.js";

/**
 * @type {import('@aws-doc-sdk-examples/lib/scenario.js').Step[]}
 */
export const deploySteps = [
  new ScenarioOutput("introduction", MESSAGES.introduction, { header: true }),
  new ScenarioInput("confirmDeployment", MESSAGES.confirmDeployment, {
    type: "confirm",
  }),
  new ScenarioAction(
    "handleConfirmDeployment",
    (c) => c.confirmDeployment === false && process.exit(),
  ),
  new ScenarioOutput(
    "creatingTable",
    MESSAGES.creatingTable.replace("${TABLE_NAME}", NAMES.tableName),
  ),
  new ScenarioAction("createTable", async () => {
    const client = new DynamoDBClient({});
    await client.send(
      new CreateTableCommand({
        TableName: NAMES.tableName,
        ProvisionedThroughput: {
          ReadCapacityUnits: 5,
          WriteCapacityUnits: 5,
        },
        AttributeDefinitions: [
          {
            AttributeName: "MediaType",
            AttributeType: "S",
          }
        ]
      })
    );
  })
];
```

```

    },
    {
      AttributeName: "ItemId",
      AttributeType: "N",
    },
  ],
  KeySchema: [
    {
      AttributeName: "MediaType",
      KeyType: "HASH",
    },
    {
      AttributeName: "ItemId",
      KeyType: "RANGE",
    },
  ],
}),
);
await waitUntilTableExists({ client }, { TableName: NAMES.tableName });
}),
new ScenarioOutput(
  "createdTable",
  MESSAGES.createdTable.replace("${TABLE_NAME}", NAMES.tableName),
),
new ScenarioOutput(
  "populatingTable",
  MESSAGES.populatingTable.replace("${TABLE_NAME}", NAMES.tableName),
),
new ScenarioAction("populateTable", () => {
  const client = new DynamoDBClient({});
  /**
   * @type {{ default: import("@aws-sdk/client-dynamodb").PutRequest['Item'][] }}
   */
  const recommendations = JSON.parse(
    readFileSync(join(RESOURCES_PATH, "recommendations.json")),
  );

  return client.send(
    new BatchWriteItemCommand({
      RequestItems: {
        [NAMES.tableName]: recommendations.map((item) => ({
          PutRequest: { Item: item },
        })),
      },
    }),
  ),
},

```

```
    }},
  );
}),
new ScenarioOutput(
  "populatedTable",
  MESSAGES.populatedTable.replace("${TABLE_NAME}", NAMES.tableName),
),
new ScenarioOutput(
  "creatingKeyPair",
  MESSAGES.creatingKeyPair.replace("${KEY_PAIR_NAME}", NAMES.keyPairName),
),
new ScenarioAction("createKeyPair", async () => {
  const client = new EC2Client({});
  const { KeyMaterial } = await client.send(
    new CreateKeyPairCommand({
      KeyName: NAMES.keyPairName,
    }),
  );
  writeFileSync(`${NAMES.keyPairName}.pem`, KeyMaterial, { mode: 0o600 });
}),
new ScenarioOutput(
  "createdKeyPair",
  MESSAGES.createdKeyPair.replace("${KEY_PAIR_NAME}", NAMES.keyPairName),
),
new ScenarioOutput(
  "creatingInstancePolicy",
  MESSAGES.creatingInstancePolicy.replace(
    "${INSTANCE_POLICY_NAME}",
    NAMES.instancePolicyName,
  ),
),
new ScenarioAction("createInstancePolicy", async (state) => {
  const client = new IAMClient({});
  const {
    Policy: { Arn },
  } = await client.send(
    new CreatePolicyCommand({
      PolicyName: NAMES.instancePolicyName,
      PolicyDocument: readFileSync(
        join(RESOURCES_PATH, "instance_policy.json"),
      ),
    }),
  );
});
```

```
    state.instancePolicyArn = Arn;
  }),
  new ScenarioOutput("createdInstancePolicy", (state) =>
    MESSAGES.createdInstancePolicy
      .replace("${INSTANCE_POLICY_NAME}", NAMES.instancePolicyName)
      .replace("${INSTANCE_POLICY_ARN}", state.instancePolicyArn),
  ),
  new ScenarioOutput(
    "creatingInstanceRole",
    MESSAGES.creatingInstanceRole.replace(
      "${INSTANCE_ROLE_NAME}",
      NAMES.instanceRoleName,
    ),
  ),
  new ScenarioAction("createInstanceRole", () => {
    const client = new IAMClient({});
    return client.send(
      new CreateRoleCommand({
        RoleName: NAMES.instanceRoleName,
        AssumeRolePolicyDocument: readFileSync(
          join(ROOT, "assume-role-policy.json"),
        ),
      }),
    ),
  }),
  new ScenarioOutput(
    "createdInstanceRole",
    MESSAGES.createdInstanceRole.replace(
      "${INSTANCE_ROLE_NAME}",
      NAMES.instanceRoleName,
    ),
  ),
  new ScenarioOutput(
    "attachingPolicyToRole",
    MESSAGES.attachingPolicyToRole
      .replace("${INSTANCE_ROLE_NAME}", NAMES.instanceRoleName)
      .replace("${INSTANCE_POLICY_NAME}", NAMES.instancePolicyName),
  ),
  new ScenarioAction("attachPolicyToRole", async (state) => {
    const client = new IAMClient({});
    await client.send(
      new AttachRolePolicyCommand({
        RoleName: NAMES.instanceRoleName,
        PolicyArn: state.instancePolicyArn,
```

```
    }),
  );
}),
new ScenarioOutput(
  "attachedPolicyToRole",
  MESSAGES.attachedPolicyToRole
    .replace("${INSTANCE_POLICY_NAME}", NAMES.instancePolicyName)
    .replace("${INSTANCE_ROLE_NAME}", NAMES.instanceRoleName),
),
new ScenarioOutput(
  "creatingInstanceProfile",
  MESSAGES.creatingInstanceProfile.replace(
    "${INSTANCE_PROFILE_NAME}",
    NAMES.instanceProfileName,
  ),
),
new ScenarioAction("createInstanceProfile", async (state) => {
  const client = new IAMClient({});
  const {
    InstanceProfile: { Arn },
  } = await client.send(
    new CreateInstanceProfileCommand({
      InstanceProfileName: NAMES.instanceProfileName,
    }),
  );
  state.instanceProfileArn = Arn;

  await waitUntilInstanceProfileExists(
    { client },
    { InstanceProfileName: NAMES.instanceProfileName },
  );
}),
new ScenarioOutput("createdInstanceProfile", (state) =>
  MESSAGES.createdInstanceProfile
    .replace("${INSTANCE_PROFILE_NAME}", NAMES.instanceProfileName)
    .replace("${INSTANCE_PROFILE_ARN}", state.instanceProfileArn),
),
new ScenarioOutput(
  "addingRoleToInstanceProfile",
  MESSAGES.addingRoleToInstanceProfile
    .replace("${INSTANCE_PROFILE_NAME}", NAMES.instanceProfileName)
    .replace("${INSTANCE_ROLE_NAME}", NAMES.instanceRoleName),
),
new ScenarioAction("addRoleToInstanceProfile", () => {
```

```

const client = new IAMClient({});
return client.send(
  new AddRoleToInstanceProfileCommand({
    RoleName: NAMES.instanceRoleName,
    InstanceProfileName: NAMES.instanceProfileName,
  }),
);
}),
new ScenarioOutput(
  "addedRoleToInstanceProfile",
  MESSAGES.addedRoleToInstanceProfile
    .replace("${INSTANCE_PROFILE_NAME}", NAMES.instanceProfileName)
    .replace("${INSTANCE_ROLE_NAME}", NAMES.instanceRoleName),
),
...initParamsSteps,
new ScenarioOutput("creatingLaunchTemplate", MESSAGES.creatingLaunchTemplate),
new ScenarioAction("createLaunchTemplate", async () => {
  const ssmClient = new SSMClient({});
  const { Parameter } = await ssmClient.send(
    new GetParameterCommand({
      Name: "/aws/service/ami-amazon-linux-latest/amzn2-ami-hvm-x86_64-gp2",
    }),
  );
  const ec2Client = new EC2Client({});
  await ec2Client.send(
    new CreateLaunchTemplateCommand({
      LaunchTemplateName: NAMES.launchTemplateName,
      LaunchTemplateData: {
        InstanceType: "t3.micro",
        ImageId: Parameter.Value,
        IamInstanceProfile: { Name: NAMES.instanceProfileName },
        UserData: readFileSync(
          join(RESOURCES_PATH, "server_startup_script.sh"),
        ).toString("base64"),
        KeyName: NAMES.keyPairName,
      },
    }),
  );
}),
new ScenarioOutput(
  "createdLaunchTemplate",
  MESSAGES.createdLaunchTemplate.replace(
    "${LAUNCH_TEMPLATE_NAME}",
    NAMES.launchTemplateName,
  ),

```

```

    ),
  ),
  new ScenarioOutput(
    "creatingAutoScalingGroup",
    MESSAGES.creatingAutoScalingGroup.replace(
      "${AUTO_SCALING_GROUP_NAME}",
      NAMES.autoScalingGroupName,
    ),
  ),
  new ScenarioAction("createAutoScalingGroup", async (state) => {
    const ec2Client = new EC2Client({});
    const { AvailabilityZones } = await ec2Client.send(
      new DescribeAvailabilityZonesCommand({}),
    );
    state.availabilityZoneNames = AvailabilityZones.map((az) => az.ZoneName);
    const autoScalingClient = new AutoScalingClient({});
    await retry({ intervalInMs: 1000, maxRetries: 30 }, () =>
      autoScalingClient.send(
        new CreateAutoScalingGroupCommand({
          AvailabilityZones: state.availabilityZoneNames,
          AutoScalingGroupName: NAMES.autoScalingGroupName,
          LaunchTemplate: {
            LaunchTemplateName: NAMES.launchTemplateName,
            Version: "$Default",
          },
          MinSize: 3,
          MaxSize: 3,
        })),
    );
  }),
  new ScenarioOutput(
    "createdAutoScalingGroup",
    /**
     * @param {{ availabilityZoneNames: string[] }} state
     */
    (state) =>
      MESSAGES.createdAutoScalingGroup
        .replace("${AUTO_SCALING_GROUP_NAME}", NAMES.autoScalingGroupName)
        .replace(
          "${AVAILABILITY_ZONE_NAMES}",
          state.availabilityZoneNames.join(", "),
        ),
  ),
),

```

```

new ScenarioInput("confirmContinue", MESSAGES.confirmContinue, {
  type: "confirm",
}),
new ScenarioOutput("loadBalancer", MESSAGES.loadBalancer),
new ScenarioOutput("gettingVpc", MESSAGES.gettingVpc),
new ScenarioAction("getVpc", async (state) => {
  const client = new EC2Client({});
  const { Vpcs } = await client.send(
    new DescribeVpcsCommand({
      Filters: [{ Name: "is-default", Values: ["true"] }],
    }),
  );
  state.defaultVpc = Vpcs[0].VpcId;
}),
new ScenarioOutput("gotVpc", (state) =>
  MESSAGES.gotVpc.replace("${VPC_ID}", state.defaultVpc),
),
new ScenarioOutput("gettingSubnets", MESSAGES.gettingSubnets),
new ScenarioAction("getSubnets", async (state) => {
  const client = new EC2Client({});
  const { Subnets } = await client.send(
    new DescribeSubnetsCommand({
      Filters: [
        { Name: "vpc-id", Values: [state.defaultVpc] },
        { Name: "availability-zone", Values: state.availabilityZoneNames },
        { Name: "default-for-az", Values: ["true"] },
      ],
    }),
  );
  state.subnets = Subnets.map((subnet) => subnet.SubnetId);
}),
new ScenarioOutput(
  "gotSubnets",
  /**
   * @param {{ subnets: string[] }} state
   */
  (state) =>
    MESSAGES.gotSubnets.replace("${SUBNETS}", state.subnets.join(", ")),
),
new ScenarioOutput(
  "creatingLoadBalancerTargetGroup",
  MESSAGES.creatingLoadBalancerTargetGroup.replace(
    "${TARGET_GROUP_NAME}",
    NAMES.loadBalancerTargetGroupName,

```



```
    ),
  ),
  new ScenarioAction("createLoadBalancerTargetGroup", async (state) => {
    const client = new ElasticLoadBalancingV2Client({});
    const { TargetGroups } = await client.send(
      new CreateTargetGroupCommand({
        Name: NAMES.loadBalancerTargetGroupName,
        Protocol: "HTTP",
        Port: 80,
        HealthCheckPath: "/healthcheck",
        HealthCheckIntervalSeconds: 10,
        HealthCheckTimeoutSeconds: 5,
        HealthyThresholdCount: 2,
        UnhealthyThresholdCount: 2,
        VpcId: state.defaultVpc,
      })
    );
    const targetGroup = TargetGroups[0];
    state.targetGroupArn = targetGroup.TargetGroupArn;
    state.targetGroupProtocol = targetGroup.Protocol;
    state.targetGroupPort = targetGroup.Port;
  }),
  new ScenarioOutput(
    "createdLoadBalancerTargetGroup",
    MESSAGES.createdLoadBalancerTargetGroup.replace(
      "${TARGET_GROUP_NAME}",
      NAMES.loadBalancerTargetGroupName,
    ),
  ),
  new ScenarioOutput(
    "creatingLoadBalancer",
    MESSAGES.creatingLoadBalancer.replace("${LB_NAME}", NAMES.loadBalancerName),
  ),
  new ScenarioAction("createLoadBalancer", async (state) => {
    const client = new ElasticLoadBalancingV2Client({});
    const { LoadBalancers } = await client.send(
      new CreateLoadBalancerCommand({
        Name: NAMES.loadBalancerName,
        Subnets: state.subnets,
      })
    );
    state.loadBalancerDns = LoadBalancers[0].DNSName;
    state.loadBalancerArn = LoadBalancers[0].LoadBalancerArn;
    await waitUntilLoadBalancerAvailable(
```

```
    { client },
    { Names: [NAMES.loadBalancerName] },
  );
}),
new ScenarioOutput("createdLoadBalancer", (state) =>
  MESSAGES.createdLoadBalancer
    .replace("${LB_NAME}", NAMES.loadBalancerName)
    .replace("${DNS_NAME}", state.loadBalancerDns),
),
new ScenarioOutput(
  "creatingListener",
  MESSAGES.creatingLoadBalancerListener
    .replace("${LB_NAME}", NAMES.loadBalancerName)
    .replace("${TARGET_GROUP_NAME}", NAMES.loadBalancerTargetGroupName),
),
new ScenarioAction("createListener", async (state) => {
  const client = new ElasticLoadBalancingV2Client({});
  const { Listeners } = await client.send(
    new CreateListenerCommand({
      LoadBalancerArn: state.loadBalancerArn,
      Protocol: state.targetGroupProtocol,
      Port: state.targetGroupPort,
      DefaultActions: [
        { Type: "forward", TargetGroupArn: state.targetGroupArn },
      ],
    })
  );
  const listener = Listeners[0];
  state.loadBalancerListenerArn = listener.ListenerArn;
}),
new ScenarioOutput("createdListener", (state) =>
  MESSAGES.createdLoadBalancerListener.replace(
    "${LB_LISTENER_ARN}",
    state.loadBalancerListenerArn,
  ),
),
new ScenarioOutput(
  "attachingLoadBalancerTargetGroup",
  MESSAGES.attachingLoadBalancerTargetGroup
    .replace("${TARGET_GROUP_NAME}", NAMES.loadBalancerTargetGroupName)
    .replace("${AUTO_SCALING_GROUP_NAME}", NAMES.autoScalingGroupName),
),
new ScenarioAction("attachLoadBalancerTargetGroup", async (state) => {
  const client = new AutoScalingClient({});
```

```

    await client.send(
      new AttachLoadBalancerTargetGroupsCommand({
        AutoScalingGroupName: NAMES.autoScalingGroupName,
        TargetGroupARNs: [state.targetGroupArn],
      }),
    );
  }),
  new ScenarioOutput(
    "attachedLoadBalancerTargetGroup",
    MESSAGES.attachedLoadBalancerTargetGroup,
  ),
  new ScenarioOutput("verifyingInboundPort", MESSAGES.verifyingInboundPort),
  new ScenarioAction(
    "verifyInboundPort",
    /**
     *
     * @param {{ defaultSecurityGroup: import('@aws-sdk/client-ec2').SecurityGroup}}
state
     */
    async (state) => {
      const client = new EC2Client({});
      const { SecurityGroups } = await client.send(
        new DescribeSecurityGroupsCommand({
          Filters: [{ Name: "group-name", Values: ["default"] }],
        }),
      );
      if (!SecurityGroups) {
        state.verifyInboundPortError = new Error(MESSAGES.noSecurityGroups);
      }
      state.defaultSecurityGroup = SecurityGroups[0];

      /**
       * @type {string}
       */
      const ipResponse = (await axios.get("http://checkip.amazonaws.com")).data;
      state.myIp = ipResponse.trim();
      const myIpRules = state.defaultSecurityGroup.IpPermissions.filter(
        ({ IpRanges }) =>
          IpRanges.some(
            ({ CidrIp }) =>
              CidrIp.startsWith(state.myIp) || CidrIp === "0.0.0.0/0",
          ),
      )
        .filter(({ IpProtocol }) => IpProtocol === "tcp")
    },
  ),

```

```
        .filter(({ FromPort }) => FromPort === 80);

        state.myIpRules = myIpRules;
    },
),
new ScenarioOutput(
    "verifiedInboundPort",
    /**
     * @param {{ myIpRules: any[] }} state
     */
    (state) => {
        if (state.myIpRules.length > 0) {
            return MESSAGES.foundIpRules.replace(
                "${IP_RULES}",
                JSON.stringify(state.myIpRules, null, 2),
            );
        } else {
            return MESSAGES.noIpRules;
        }
    },
),
new ScenarioInput(
    "shouldAddInboundRule",
    /**
     * @param {{ myIpRules: any[] }} state
     */
    (state) => {
        if (state.myIpRules.length > 0) {
            return false;
        } else {
            return MESSAGES.noIpRules;
        }
    },
    { type: "confirm" },
),
new ScenarioAction(
    "addInboundRule",
    /**
     * @param {{ defaultSecurityGroup: import('@aws-sdk/client-ec2').SecurityGroup }} state
     */
    async (state) => {
        if (!state.shouldAddInboundRule) {
            return;
        }
    }
);
```

```
    }

    const client = new EC2Client({});
    await client.send(
      new AuthorizeSecurityGroupIngressCommand({
        GroupId: state.defaultSecurityGroup.GroupId,
        CidrIp: `${state.myIp}/32`,
        FromPort: 80,
        ToPort: 80,
        IpProtocol: "tcp",
      }),
    );
  },
),
new ScenarioOutput("addedInboundRule", (state) => {
  if (state.shouldAddInboundRule) {
    return MESSAGES.addedInboundRule.replace("${IP_ADDRESS}", state.myIp);
  } else {
    return false;
  }
}),
new ScenarioOutput("verifyingEndpoint", (state) =>
  MESSAGES.verifyingEndpoint.replace("${DNS_NAME}", state.loadBalancerDns),
),
new ScenarioAction("verifyEndpoint", async (state) => {
  try {
    const response = await retry({ intervalInMs: 2000, maxRetries: 30 }, () =>
      axios.get(`http://${state.loadBalancerDns}`),
    );
    state.endpointResponse = JSON.stringify(response.data, null, 2);
  } catch (e) {
    state.verifyEndpointError = e;
  }
}),
new ScenarioOutput("verifiedEndpoint", (state) => {
  if (state.verifyEndpointError) {
    console.error(state.verifyEndpointError);
  } else {
    return MESSAGES.verifiedEndpoint.replace(
      "${ENDPOINT_RESPONSE}",
      state.endpointResponse,
    );
  }
}),
}),
```

```
    saveState,  
  ];
```

Create steps to run the demo.

```
import { readFileSync } from "node:fs";  
import { join } from "node:path";  
  
import axios from "axios";  
  
import {  
  DescribeTargetGroupsCommand,  
  DescribeTargetHealthCommand,  
  ElasticLoadBalancingV2Client,  
} from "@aws-sdk/client-elastic-load-balancing-v2";  
import {  
  DescribeInstanceInformationCommand,  
  PutParameterCommand,  
  SSMClient,  
  SendCommandCommand,  
} from "@aws-sdk/client-ssm";  
import {  
  IAMClient,  
  CreatePolicyCommand,  
  CreateRoleCommand,  
  AttachRolePolicyCommand,  
  CreateInstanceProfileCommand,  
  AddRoleToInstanceProfileCommand,  
  waitUntilInstanceProfileExists,  
} from "@aws-sdk/client-iam";  
import {  
  AutoScalingClient,  
  DescribeAutoScalingGroupsCommand,  
  TerminateInstanceInAutoScalingGroupCommand,  
} from "@aws-sdk/client-auto-scaling";  
import {  
  DescribeIamInstanceProfileAssociationsCommand,  
  EC2Client,  
  RebootInstancesCommand,  
  ReplaceIamInstanceProfileAssociationCommand,  
} from "@aws-sdk/client-ec2";
```

```
import {
  ScenarioAction,
  ScenarioInput,
  ScenarioOutput,
} from "@aws-doc-sdk-examples/lib/scenario/scenario.js";
import { retry } from "@aws-doc-sdk-examples/lib/utils/util-timers.js";

import { MESSAGES, NAMES, RESOURCES_PATH } from "./constants.js";
import { findLoadBalancer } from "./shared.js";

const getRecommendation = new ScenarioAction(
  "getRecommendation",
  async (state) => {
    const loadBalancer = await findLoadBalancer(NAMES.loadBalancerName);
    if (loadBalancer) {
      state.loadBalancerDnsName = loadBalancer.DNSName;
      try {
        state.recommendation = (
          await axios.get(`http://${state.loadBalancerDnsName}`)
        ).data;
      } catch (e) {
        state.recommendation = e instanceof Error ? e.message : e;
      }
    } else {
      throw new Error(MESSAGES.demoFindLoadBalancerError);
    }
  },
);

const getRecommendationResult = new ScenarioOutput(
  "getRecommendationResult",
  (state) =>
    `Recommendation:\n${JSON.stringify(state.recommendation, null, 2)}`,
  { preformatted: true },
);

const getHealthCheck = new ScenarioAction("getHealthCheck", async (state) => {
  const client = new ElasticLoadBalancingV2Client({});
  const { TargetGroups } = await client.send(
    new DescribeTargetGroupsCommand({
      Names: [NAMES.loadBalancerTargetGroupName],
    }),
  );
});
```

```

const { TargetHealthDescriptions } = await client.send(
  new DescribeTargetHealthCommand({
    TargetGroupArn: TargetGroups[0].TargetGroupArn,
  }),
);
state.targetHealthDescriptions = TargetHealthDescriptions;
});

const getHealthCheckResult = new ScenarioOutput(
  "getHealthCheckResult",
  /**
   * @param {{ targetHealthDescriptions: import('@aws-sdk/client-elastic-load-
balancing-v2').TargetHealthDescription[] }} state
   */
  (state) => {
    const status = state.targetHealthDescriptions
      .map((th) => `${th.Target.Id}: ${th.TargetHealth.State}`)
      .join("\n");
    return `Health check:\n${status}`;
  },
  { preformatted: true },
);

const loadBalancerLoop = new ScenarioAction(
  "loadBalancerLoop",
  getRecommendation.action,
  {
    whileConfig: {
      whileFn: ({ loadBalancerCheck }) => loadBalancerCheck,
      input: new ScenarioInput(
        "loadBalancerCheck",
        MESSAGES.demoLoadBalancerCheck,
        {
          type: "confirm",
        },
      ),
      output: getRecommendationResult,
    },
  },
);

const healthCheckLoop = new ScenarioAction(
  "healthCheckLoop",
  getHealthCheck.action,

```



```
{
  whileConfig: {
    whileFn: ({ healthCheck }) => healthCheck,
    input: new ScenarioInput("healthCheck", MESSAGES.demoHealthCheck, {
      type: "confirm",
    }),
    output: getHealthCheckResult,
  },
},
);

const statusSteps = [
  getRecommendation,
  getRecommendationResult,
  getHealthCheck,
  getHealthCheckResult,
];

/**
 * @type {import('@aws-doc-sdk-examples/lib/scenario.js').Step[]}
 */
export const demoSteps = [
  new ScenarioOutput("header", MESSAGES.demoHeader, { header: true }),
  new ScenarioOutput("sanityCheck", MESSAGES.demoSanityCheck),
  ...statusSteps,
  new ScenarioInput(
    "brokenDependencyConfirmation",
    MESSAGES.demoBrokenDependencyConfirmation,
    { type: "confirm" },
  ),
  new ScenarioAction("brokenDependency", async (state) => {
    if (!state.brokenDependencyConfirmation) {
      process.exit();
    } else {
      const client = new SSMClient({});
      state.badTableName = `fake-table-${Date.now()}`;
      await client.send(
        new PutParameterCommand({
          Name: NAMES.ssmTableNameKey,
          Value: state.badTableName,
          Overwrite: true,
          Type: "String",
        })
      ),
    }
  });
];
```

```
    }
  }},
  new ScenarioOutput("testBrokenDependency", (state) =>
    MESSAGES.demoTestBrokenDependency.replace(
      "${TABLE_NAME}",
      state.badTableName,
    ),
  ),
  ...statusSteps,
  new ScenarioInput(
    "staticResponseConfirmation",
    MESSAGES.demoStaticResponseConfirmation,
    { type: "confirm" },
  ),
  new ScenarioAction("staticResponse", async (state) => {
    if (!state.staticResponseConfirmation) {
      process.exit();
    } else {
      const client = new SSMClient({});
      await client.send(
        new PutParameterCommand({
          Name: NAMES.ssmFailureResponseKey,
          Value: "static",
          Overwrite: true,
          Type: "String",
        })),
      );
    }
  })),
  new ScenarioOutput("testStaticResponse", MESSAGES.demoTestStaticResponse),
  ...statusSteps,
  new ScenarioInput(
    "badCredentialsConfirmation",
    MESSAGES.demoBadCredentialsConfirmation,
    { type: "confirm" },
  ),
  new ScenarioAction("badCredentialsExit", (state) => {
    if (!state.badCredentialsConfirmation) {
      process.exit();
    }
  })),
  new ScenarioAction("fixDynamoDBName", async () => {
    const client = new SSMClient({});
    await client.send(
```

```

    new PutParameterCommand({
      Name: NAMES.ssmTableNameKey,
      Value: NAMES.tableName,
      Overwrite: true,
      Type: "String",
    }),
  );
}),
new ScenarioAction(
  "badCredentials",
  /**
   * @param {{ targetInstance: import('@aws-sdk/client-auto-scaling').Instance }}
state
  */
  async (state) => {
    await createSsmOnlyInstanceProfile();
    const autoScalingClient = new AutoScalingClient({});
    const { AutoScalingGroups } = await autoScalingClient.send(
      new DescribeAutoScalingGroupsCommand({
        AutoScalingGroupNames: [NAMES.autoScalingGroupName],
      }),
    );
    state.targetInstance = AutoScalingGroups[0].Instances[0];
    const ec2Client = new EC2Client({});
    const { IamInstanceProfileAssociations } = await ec2Client.send(
      new DescribeIamInstanceProfileAssociationsCommand({
        Filters: [
          { Name: "instance-id", Values: [state.targetInstance.InstanceId] },
        ],
      }),
    );
    state.instanceProfileAssociationId =
      IamInstanceProfileAssociations[0].AssociationId;
    await retry({ intervalInMs: 1000, maxRetries: 30 }, () =>
      ec2Client.send(
        new ReplaceIamInstanceProfileAssociationCommand({
          AssociationId: state.instanceProfileAssociationId,
          IamInstanceProfile: { Name: NAMES.ssmOnlyInstanceProfileName },
        }),
      ),
    );

    await ec2Client.send(
      new RebootInstancesCommand({

```

```
        InstanceIds: [state.targetInstance.InstanceId],
    })),
  );

  const ssmClient = new SSMClient({});
  await retry({ intervalInMs: 20000, maxRetries: 15 }, async () => {
    const { InstanceInformationList } = await ssmClient.send(
      new DescribeInstanceInformationCommand({}),
    );

    const instance = InstanceInformationList.find(
      (info) => info.InstanceId === state.targetInstance.InstanceId,
    );

    if (!instance) {
      throw new Error("Instance not found.");
    }
  });

  await ssmClient.send(
    new SendCommandCommand({
      InstanceIds: [state.targetInstance.InstanceId],
      DocumentName: "AWS-RunShellScript",
      Parameters: { commands: ["cd / && sudo python3 server.py 80"] },
    }),
  );
},
),
new ScenarioOutput(
  "testBadCredentials",
  /**
   * @param {{ targetInstance: import('@aws-sdk/client-ssm').InstanceInformation}}
state
  */
  (state) =>
    MESSAGES.demoTestBadCredentials.replace(
      "${INSTANCE_ID}",
      state.targetInstance.InstanceId,
    ),
),
loadBalancerLoop,
new ScenarioInput(
  "deepHealthCheckConfirmation",
  MESSAGES.demoDeepHealthCheckConfirmation,
```

```

    { type: "confirm" },
  ),
  new ScenarioAction("deepHealthCheckExit", (state) => {
    if (!state.deepHealthCheckConfirmation) {
      process.exit();
    }
  }),
  new ScenarioAction("deepHealthCheck", async () => {
    const client = new SSMClient({});
    await client.send(
      new PutParameterCommand({
        Name: NAMES.ssmHealthCheckKey,
        Value: "deep",
        Overwrite: true,
        Type: "String",
      })
    );
  }),
  new ScenarioOutput("testDeepHealthCheck", MESSAGES.demoTestDeepHealthCheck),
  healthCheckLoop,
  loadBalancerLoop,
  new ScenarioInput(
    "killInstanceConfirmation",
    /**
     * @param {{ targetInstance: import('@aws-sdk/client-ssm').InstanceInformation }} state
     */
    (state) =>
      MESSAGES.demoKillInstanceConfirmation.replace(
        "${INSTANCE_ID}",
        state.targetInstance.InstanceId,
      ),
    { type: "confirm" },
  ),
  new ScenarioAction("killInstanceExit", (state) => {
    if (!state.killInstanceConfirmation) {
      process.exit();
    }
  }),
  new ScenarioAction(
    "killInstance",
    /**
     * @param {{ targetInstance: import('@aws-sdk/client-ssm').InstanceInformation }} state

```

```
    */
    async (state) => {
      const client = new AutoScalingClient({});
      await client.send(
        new TerminateInstanceInAutoScalingGroupCommand({
          InstanceId: state.targetInstance.InstanceId,
          ShouldDecrementDesiredCapacity: false,
        }),
      );
    },
  ),
  new ScenarioOutput("testKillInstance", MESSAGES.demoTestKillInstance),
  healthCheckLoop,
  loadBalancerLoop,
  new ScenarioInput("failOpenConfirmation", MESSAGES.demoFailOpenConfirmation, {
    type: "confirm",
  }),
  new ScenarioAction("failOpenExit", (state) => {
    if (!state.failOpenConfirmation) {
      process.exit();
    }
  }),
  new ScenarioAction("failOpen", () => {
    const client = new SSMClient({});
    return client.send(
      new PutParameterCommand({
        Name: NAMES.ssmTableNameKey,
        Value: `fake-table-${Date.now()}`,
        Overwrite: true,
        Type: "String",
      }),
    );
  }),
  new ScenarioOutput("testFailOpen", MESSAGES.demoFailOpenTest),
  healthCheckLoop,
  loadBalancerLoop,
  new ScenarioInput(
    "resetTableConfirmation",
    MESSAGES.demoResetTableConfirmation,
    { type: "confirm" },
  ),
  new ScenarioAction("resetTableExit", (state) => {
    if (!state.resetTableConfirmation) {
      process.exit();
    }
  })
}
```

```
    }
  }),
  new ScenarioAction("resetTable", async () => {
    const client = new SSMClient({});
    await client.send(
      new PutParameterCommand({
        Name: NAMES.ssmTableNameKey,
        Value: NAMES.tableName,
        Overwrite: true,
        Type: "String",
      }),
    );
  }),
  new ScenarioOutput("testResetTable", MESSAGES.demoTestResetTable),
  healthCheckLoop,
  loadBalancerLoop,
];

async function createSsmOnlyInstanceProfile() {
  const iamClient = new IAMClient({});
  const { Policy } = await iamClient.send(
    new CreatePolicyCommand({
      PolicyName: NAMES.ssmOnlyPolicyName,
      PolicyDocument: readFileSync(
        join(RESOURCES_PATH, "ssm_only_policy.json"),
      ),
    }),
  );
  await iamClient.send(
    new CreateRoleCommand({
      RoleName: NAMES.ssmOnlyRoleName,
      AssumeRolePolicyDocument: JSON.stringify({
        Version: "2012-10-17",
        Statement: [
          {
            Effect: "Allow",
            Principal: { Service: "ec2.amazonaws.com" },
            Action: "sts:AssumeRole",
          },
        ],
      }),
    ),
  );
  await iamClient.send(
```

```
    new AttachRolePolicyCommand({
      RoleName: NAMES.ssmOnlyRoleName,
      PolicyArn: Policy.Arn,
    }),
  );
await iamClient.send(
  new AttachRolePolicyCommand({
    RoleName: NAMES.ssmOnlyRoleName,
    PolicyArn: "arn:aws:iam::aws:policy/AmazonSSMManagedInstanceCore",
  }),
);
const { InstanceProfile } = await iamClient.send(
  new CreateInstanceProfileCommand({
    InstanceProfileName: NAMES.ssmOnlyInstanceProfileName,
  }),
);
await waitUntilInstanceProfileExists(
  { client: iamClient },
  { InstanceProfileName: NAMES.ssmOnlyInstanceProfileName },
);
await iamClient.send(
  new AddRoleToInstanceProfileCommand({
    InstanceProfileName: NAMES.ssmOnlyInstanceProfileName,
    RoleName: NAMES.ssmOnlyRoleName,
  }),
);

return InstanceProfile;
}
```

Create steps to destroy all of the resources.

```
import { unlinkSync } from "node:fs";

import { DynamoDBClient, DeleteTableCommand } from "@aws-sdk/client-dynamodb";
import {
  EC2Client,
  DeleteKeyPairCommand,
  DeleteLaunchTemplateCommand,
  RevokeSecurityGroupIngressCommand,
} from "@aws-sdk/client-ec2";
import {
```



```

    IAMClient,
    DeleteInstanceProfileCommand,
    RemoveRoleFromInstanceProfileCommand,
    DeletePolicyCommand,
    DeleteRoleCommand,
    DetachRolePolicyCommand,
    paginateListPolicies,
} from "@aws-sdk/client-iam";
import {
    AutoScalingClient,
    DeleteAutoScalingGroupCommand,
    TerminateInstanceInAutoScalingGroupCommand,
    UpdateAutoScalingGroupCommand,
    paginateDescribeAutoScalingGroups,
} from "@aws-sdk/client-auto-scaling";
import {
    DeleteLoadBalancerCommand,
    DeleteTargetGroupCommand,
    DescribeTargetGroupsCommand,
    ElasticLoadBalancingV2Client,
} from "@aws-sdk/client-elastic-load-balancing-v2";

import {
    ScenarioOutput,
    ScenarioInput,
    ScenarioAction,
} from "@aws-doc-sdk-examples/lib/scenario/index.js";
import { loadState } from "@aws-doc-sdk-examples/lib/scenario/steps-common.js";
import { retry } from "@aws-doc-sdk-examples/lib/utils/util-timers.js";

import { MESSAGES, NAMES } from "./constants.js";
import { findLoadBalancer } from "./shared.js";

/**
 * @type {import('@aws-doc-sdk-examples/lib/scenario.js').Step[]}
 */
export const destroySteps = [
    loadState,
    new ScenarioInput("destroy", MESSAGES.destroy, { type: "confirm" }),
    new ScenarioAction(
        "abort",
        (state) => state.destroy === false && process.exit(),
    ),
    new ScenarioAction("deleteTable", async (c) => {

```

```
    try {
      const client = new DynamoDBClient({});
      await client.send(new DeleteTableCommand({ TableName: NAMES.tableName }));
    } catch (e) {
      c.deleteTableError = e;
    }
  })),
  new ScenarioOutput("deleteTableResult", (state) => {
    if (state.deleteTableError) {
      console.error(state.deleteTableError);
      return MESSAGES.deleteTableError.replace(
        "${TABLE_NAME}",
        NAMES.tableName,
      );
    } else {
      return MESSAGES.deletedTable.replace("${TABLE_NAME}", NAMES.tableName);
    }
  })),
  new ScenarioAction("deleteKeyPair", async (state) => {
    try {
      const client = new EC2Client({});
      await client.send(
        new DeleteKeyPairCommand({ KeyName: NAMES.keyPairName }),
      );
      unlinkSync(`${NAMES.keyPairName}.pem`);
    } catch (e) {
      state.deleteKeyPairError = e;
    }
  })),
  new ScenarioOutput("deleteKeyPairResult", (state) => {
    if (state.deleteKeyPairError) {
      console.error(state.deleteKeyPairError);
      return MESSAGES.deleteKeyPairError.replace(
        "${KEY_PAIR_NAME}",
        NAMES.keyPairName,
      );
    } else {
      return MESSAGES.deletedKeyPair.replace(
        "${KEY_PAIR_NAME}",
        NAMES.keyPairName,
      );
    }
  })),
  new ScenarioAction("detachPolicyFromRole", async (state) => {
```

```
try {
  const client = new IAMClient({});
  const policy = await findPolicy(NAMES.instancePolicyName);

  if (!policy) {
    state.detachPolicyFromRoleError = new Error(
      `Policy ${NAMES.instancePolicyName} not found.`
    );
  } else {
    await client.send(
      new DetachRolePolicyCommand({
        RoleName: NAMES.instanceRoleName,
        PolicyArn: policy.Arn,
      }),
    );
  }
} catch (e) {
  state.detachPolicyFromRoleError = e;
}
}),
new ScenarioOutput("detachedPolicyFromRole", (state) => {
  if (state.detachPolicyFromRoleError) {
    console.error(state.detachPolicyFromRoleError);
    return MESSAGES.detachPolicyFromRoleError
      .replace("${INSTANCE_POLICY_NAME}", NAMES.instancePolicyName)
      .replace("${INSTANCE_ROLE_NAME}", NAMES.instanceRoleName);
  } else {
    return MESSAGES.detachedPolicyFromRole
      .replace("${INSTANCE_POLICY_NAME}", NAMES.instancePolicyName)
      .replace("${INSTANCE_ROLE_NAME}", NAMES.instanceRoleName);
  }
}),
new ScenarioAction("deleteInstancePolicy", async (state) => {
  const client = new IAMClient({});
  const policy = await findPolicy(NAMES.instancePolicyName);

  if (!policy) {
    state.deletePolicyError = new Error(
      `Policy ${NAMES.instancePolicyName} not found.`
    );
  } else {
    return client.send(
      new DeletePolicyCommand({
        PolicyArn: policy.Arn,
```

```
    }),
  );
}
}),
new ScenarioOutput("deletePolicyResult", (state) => {
  if (state.deletePolicyError) {
    console.error(state.deletePolicyError);
    return MESSAGES.deletePolicyError.replace(
      "${INSTANCE_POLICY_NAME}",
      NAMES.instancePolicyName,
    );
  } else {
    return MESSAGES.deletedPolicy.replace(
      "${INSTANCE_POLICY_NAME}",
      NAMES.instancePolicyName,
    );
  }
}),
new ScenarioAction("removeRoleFromInstanceProfile", async (state) => {
  try {
    const client = new IAMClient({});
    await client.send(
      new RemoveRoleFromInstanceProfileCommand({
        RoleName: NAMES.instanceRoleName,
        InstanceProfileName: NAMES.instanceProfileName,
      }),
    );
  } catch (e) {
    state.removeRoleFromInstanceProfileError = e;
  }
}),
new ScenarioOutput("removeRoleFromInstanceProfileResult", (state) => {
  if (state.removeRoleFromInstanceProfile) {
    console.error(state.removeRoleFromInstanceProfileError);
    return MESSAGES.removeRoleFromInstanceProfileError
      .replace("${INSTANCE_PROFILE_NAME}", NAMES.instanceProfileName)
      .replace("${INSTANCE_ROLE_NAME}", NAMES.instanceRoleName);
  } else {
    return MESSAGES.removedRoleFromInstanceProfile
      .replace("${INSTANCE_PROFILE_NAME}", NAMES.instanceProfileName)
      .replace("${INSTANCE_ROLE_NAME}", NAMES.instanceRoleName);
  }
}),
new ScenarioAction("deleteInstanceRole", async (state) => {
```

```
    try {
      const client = new IAMClient({});
      await client.send(
        new DeleteRoleCommand({
          RoleName: NAMES.instanceRoleName,
        }),
      );
    } catch (e) {
      state.deleteInstanceRoleError = e;
    }
  })),
  new ScenarioOutput("deleteInstanceRoleResult", (state) => {
    if (state.deleteInstanceRoleError) {
      console.error(state.deleteInstanceRoleError);
      return MESSAGES.deleteInstanceRoleError.replace(
        "${INSTANCE_ROLE_NAME}",
        NAMES.instanceRoleName,
      );
    } else {
      return MESSAGES.deletedInstanceRole.replace(
        "${INSTANCE_ROLE_NAME}",
        NAMES.instanceRoleName,
      );
    }
  })),
  new ScenarioAction("deleteInstanceProfile", async (state) => {
    try {
      const client = new IAMClient({});
      await client.send(
        new DeleteInstanceProfileCommand({
          InstanceProfileName: NAMES.instanceProfileName,
        }),
      );
    } catch (e) {
      state.deleteInstanceProfileError = e;
    }
  })),
  new ScenarioOutput("deleteInstanceProfileResult", (state) => {
    if (state.deleteInstanceProfileError) {
      console.error(state.deleteInstanceProfileError);
      return MESSAGES.deleteInstanceProfileError.replace(
        "${INSTANCE_PROFILE_NAME}",
        NAMES.instanceProfileName,
      );
    }
  }));
```

```
    } else {
      return MESSAGES.deletedInstanceProfile.replace(
        "${INSTANCE_PROFILE_NAME}",
        NAMES.instanceProfileName,
      );
    }
  })),
  new ScenarioAction("deleteAutoScalingGroup", async (state) => {
    try {
      await terminateGroupInstances(NAMES.autoScalingGroupName);
      await retry({ intervalInMs: 60000, maxRetries: 60 }, async () => {
        await deleteAutoScalingGroup(NAMES.autoScalingGroupName);
      });
    } catch (e) {
      state.deleteAutoScalingGroupError = e;
    }
  })),
  new ScenarioOutput("deleteAutoScalingGroupResult", (state) => {
    if (state.deleteAutoScalingGroupError) {
      console.error(state.deleteAutoScalingGroupError);
      return MESSAGES.deleteAutoScalingGroupError.replace(
        "${AUTO_SCALING_GROUP_NAME}",
        NAMES.autoScalingGroupName,
      );
    } else {
      return MESSAGES.deletedAutoScalingGroup.replace(
        "${AUTO_SCALING_GROUP_NAME}",
        NAMES.autoScalingGroupName,
      );
    }
  })),
  new ScenarioAction("deleteLaunchTemplate", async (state) => {
    const client = new EC2Client({});
    try {
      await client.send(
        new DeleteLaunchTemplateCommand({
          LaunchTemplateName: NAMES.launchTemplateName,
        }),
      );
    } catch (e) {
      state.deleteLaunchTemplateError = e;
    }
  })),
  new ScenarioOutput("deleteLaunchTemplateResult", (state) => {
```

```
    if (state.deleteLaunchTemplateError) {
      console.error(state.deleteLaunchTemplateError);
      return MESSAGES.deleteLaunchTemplateError.replace(
        "${LAUNCH_TEMPLATE_NAME}",
        NAMES.launchTemplateName,
      );
    } else {
      return MESSAGES.deletedLaunchTemplate.replace(
        "${LAUNCH_TEMPLATE_NAME}",
        NAMES.launchTemplateName,
      );
    }
  })),
  new ScenarioAction("deleteLoadBalancer", async (state) => {
    try {
      const client = new ElasticLoadBalancingV2Client({});
      const loadBalancer = await findLoadBalancer(NAMES.loadBalancerName);
      await client.send(
        new DeleteLoadBalancerCommand({
          LoadBalancerArn: loadBalancer.LoadBalancerArn,
        }),
      );
      await retry({ intervalInMs: 1000, maxRetries: 60 }, async () => {
        const lb = await findLoadBalancer(NAMES.loadBalancerName);
        if (lb) {
          throw new Error("Load balancer still exists.");
        }
      });
    } catch (e) {
      state.deleteLoadBalancerError = e;
    }
  })),
  new ScenarioOutput("deleteLoadBalancerResult", (state) => {
    if (state.deleteLoadBalancerError) {
      console.error(state.deleteLoadBalancerError);
      return MESSAGES.deleteLoadBalancerError.replace(
        "${LB_NAME}",
        NAMES.loadBalancerName,
      );
    } else {
      return MESSAGES.deletedLoadBalancer.replace(
        "${LB_NAME}",
        NAMES.loadBalancerName,
      );
    }
  })
```

```

    }
  }},
  new ScenarioAction("deleteLoadBalancerTargetGroup", async (state) => {
    const client = new ElasticLoadBalancingV2Client({});
    try {
      const { TargetGroups } = await client.send(
        new DescribeTargetGroupsCommand({
          Names: [NAMES.loadBalancerTargetGroupName],
        }),
      );

      await retry({ intervalInMs: 1000, maxRetries: 30 }, () =>
        client.send(
          new DeleteTargetGroupCommand({
            TargetGroupArn: TargetGroups[0].TargetGroupArn,
          }),
        ),
      );
    } catch (e) {
      state.deleteLoadBalancerTargetGroupError = e;
    }
  }},
  new ScenarioOutput("deleteLoadBalancerTargetGroupResult", (state) => {
    if (state.deleteLoadBalancerTargetGroupError) {
      console.error(state.deleteLoadBalancerTargetGroupError);
      return MESSAGES.deleteLoadBalancerTargetGroupError.replace(
        "${TARGET_GROUP_NAME}",
        NAMES.loadBalancerTargetGroupName,
      );
    } else {
      return MESSAGES.deletedLoadBalancerTargetGroup.replace(
        "${TARGET_GROUP_NAME}",
        NAMES.loadBalancerTargetGroupName,
      );
    }
  }},
  new ScenarioAction("detachSsmOnlyRoleFromProfile", async (state) => {
    try {
      const client = new IAMClient({});
      await client.send(
        new RemoveRoleFromInstanceProfileCommand({
          InstanceProfileName: NAMES.ssmOnlyInstanceProfileName,
          RoleName: NAMES.ssmOnlyRoleName,
        }),
      );
    }
  }},

```



```
    );
  } catch (e) {
    state.detachSsmOnlyRoleFromProfileError = e;
  }
}),
new ScenarioOutput("detachSsmOnlyRoleFromProfileResult", (state) => {
  if (state.detachSsmOnlyRoleFromProfileError) {
    console.error(state.detachSsmOnlyRoleFromProfileError);
    return MESSAGES.detachSsmOnlyRoleFromProfileError
      .replace("${ROLE_NAME}", NAMES.ssmOnlyRoleName)
      .replace("${PROFILE_NAME}", NAMES.ssmOnlyInstanceProfileName);
  } else {
    return MESSAGES.detachedSsmOnlyRoleFromProfile
      .replace("${ROLE_NAME}", NAMES.ssmOnlyRoleName)
      .replace("${PROFILE_NAME}", NAMES.ssmOnlyInstanceProfileName);
  }
}),
new ScenarioAction("detachSsmOnlyCustomRolePolicy", async (state) => {
  try {
    const iamClient = new IAMClient({});
    const ssmOnlyPolicy = await findPolicy(NAMES.ssmOnlyPolicyName);
    await iamClient.send(
      new DetachRolePolicyCommand({
        RoleName: NAMES.ssmOnlyRoleName,
        PolicyArn: ssmOnlyPolicy.Arn,
      }),
    );
  } catch (e) {
    state.detachSsmOnlyCustomRolePolicyError = e;
  }
}),
new ScenarioOutput("detachSsmOnlyCustomRolePolicyResult", (state) => {
  if (state.detachSsmOnlyCustomRolePolicyError) {
    console.error(state.detachSsmOnlyCustomRolePolicyError);
    return MESSAGES.detachSsmOnlyCustomRolePolicyError
      .replace("${ROLE_NAME}", NAMES.ssmOnlyRoleName)
      .replace("${POLICY_NAME}", NAMES.ssmOnlyPolicyName);
  } else {
    return MESSAGES.detachedSsmOnlyCustomRolePolicy
      .replace("${ROLE_NAME}", NAMES.ssmOnlyRoleName)
      .replace("${POLICY_NAME}", NAMES.ssmOnlyPolicyName);
  }
}),
new ScenarioAction("detachSsmOnlyAWSRolePolicy", async (state) => {
```

```
try {
  const iamClient = new IAMClient({});
  await iamClient.send(
    new DetachRolePolicyCommand({
      RoleName: NAMES.ssmOnlyRoleName,
      PolicyArn: "arn:aws:iam::aws:policy/AmazonSSMManagedInstanceCore",
    }),
  );
} catch (e) {
  state.detachSsmOnlyAWSRolePolicyError = e;
}
}),
new ScenarioOutput("detachSsmOnlyAWSRolePolicyResult", (state) => {
  if (state.detachSsmOnlyAWSRolePolicyError) {
    console.error(state.detachSsmOnlyAWSRolePolicyError);
    return MESSAGES.detachSsmOnlyAWSRolePolicyError
      .replace("${ROLE_NAME}", NAMES.ssmOnlyRoleName)
      .replace("${POLICY_NAME}", "AmazonSSMManagedInstanceCore");
  } else {
    return MESSAGES.detachedSsmOnlyAWSRolePolicy
      .replace("${ROLE_NAME}", NAMES.ssmOnlyRoleName)
      .replace("${POLICY_NAME}", "AmazonSSMManagedInstanceCore");
  }
}),
new ScenarioAction("deleteSsmOnlyInstanceProfile", async (state) => {
  try {
    const iamClient = new IAMClient({});
    await iamClient.send(
      new DeleteInstanceProfileCommand({
        InstanceProfileName: NAMES.ssmOnlyInstanceProfileName,
      }),
    );
  } catch (e) {
    state.deleteSsmOnlyInstanceProfileError = e;
  }
}),
new ScenarioOutput("deleteSsmOnlyInstanceProfileResult", (state) => {
  if (state.deleteSsmOnlyInstanceProfileError) {
    console.error(state.deleteSsmOnlyInstanceProfileError);
    return MESSAGES.deleteSsmOnlyInstanceProfileError.replace(
      "${INSTANCE_PROFILE_NAME}",
      NAMES.ssmOnlyInstanceProfileName,
    );
  } else {
```

```
    return MESSAGES.deletedSsmOnlyInstanceProfile.replace(
      "${INSTANCE_PROFILE_NAME}",
      NAMES.ssmOnlyInstanceProfileName,
    );
  }
}),
new ScenarioAction("deleteSsmOnlyPolicy", async (state) => {
  try {
    const iamClient = new IAMClient({});
    const ssmOnlyPolicy = await findPolicy(NAMES.ssmOnlyPolicyName);
    await iamClient.send(
      new DeletePolicyCommand({
        PolicyArn: ssmOnlyPolicy.Arn,
      }),
    );
  } catch (e) {
    state.deleteSsmOnlyPolicyError = e;
  }
}),
new ScenarioOutput("deleteSsmOnlyPolicyResult", (state) => {
  if (state.deleteSsmOnlyPolicyError) {
    console.error(state.deleteSsmOnlyPolicyError);
    return MESSAGES.deleteSsmOnlyPolicyError.replace(
      "${POLICY_NAME}",
      NAMES.ssmOnlyPolicyName,
    );
  } else {
    return MESSAGES.deletedSsmOnlyPolicy.replace(
      "${POLICY_NAME}",
      NAMES.ssmOnlyPolicyName,
    );
  }
}),
new ScenarioAction("deleteSsmOnlyRole", async (state) => {
  try {
    const iamClient = new IAMClient({});
    await iamClient.send(
      new DeleteRoleCommand({
        RoleName: NAMES.ssmOnlyRoleName,
      }),
    );
  } catch (e) {
    state.deleteSsmOnlyRoleError = e;
  }
})
```

```

    }),
    new ScenarioOutput("deleteSsmOnlyRoleResult", (state) => {
      if (state.deleteSsmOnlyRoleError) {
        console.error(state.deleteSsmOnlyRoleError);
        return MESSAGES.deleteSsmOnlyRoleError.replace(
          "${ROLE_NAME}",
          NAMES.ssmOnlyRoleName,
        );
      } else {
        return MESSAGES.deletedSsmOnlyRole.replace(
          "${ROLE_NAME}",
          NAMES.ssmOnlyRoleName,
        );
      }
    })),
    new ScenarioAction(
      "revokeSecurityGroupIngress",
      async (
        /** @type {{ myIp: string, defaultSecurityGroup: { GroupId: string } }} */
        state,
      ) => {
        const ec2Client = new EC2Client({});

        try {
          await ec2Client.send(
            new RevokeSecurityGroupIngressCommand({
              GroupId: state.defaultSecurityGroup.GroupId,
              CidrIp: `${state.myIp}/32`,
              FromPort: 80,
              ToPort: 80,
              IpProtocol: "tcp",
            }),
          );
        } catch (e) {
          state.revokeSecurityGroupIngressError = e;
        }
      },
    ),
    new ScenarioOutput("revokeSecurityGroupIngressResult", (state) => {
      if (state.revokeSecurityGroupIngressError) {
        console.error(state.revokeSecurityGroupIngressError);
        return MESSAGES.revokeSecurityGroupIngressError.replace(
          "${IP}",
          state.myIp,
        );
      }
    })),
  ],
);

```

```
    );
  } else {
    return MESSAGES.revokedSecurityGroupIngress.replace("${IP}", state.myIp);
  }
}),
];

/**
 * @param {string} policyName
 */
async function findPolicy(policyName) {
  const client = new IAMClient({});
  const paginatedPolicies = paginateListPolicies({ client }, {});
  for await (const page of paginatedPolicies) {
    const policy = page.Policies.find((p) => p.PolicyName === policyName);
    if (policy) {
      return policy;
    }
  }
}

/**
 * @param {string} groupName
 */
async function deleteAutoScalingGroup(groupName) {
  const client = new AutoScalingClient({});
  try {
    await client.send(
      new DeleteAutoScalingGroupCommand({
        AutoScalingGroupName: groupName,
      }),
    );
  } catch (err) {
    if (!(err instanceof Error)) {
      throw err;
    } else {
      console.log(err.name);
      throw err;
    }
  }
}

/**
 * @param {string} groupName
```

```
*/
async function terminateGroupInstances(groupName) {
  const autoScalingClient = new AutoScalingClient({});
  const group = await findAutoScalingGroup(groupName);
  await autoScalingClient.send(
    new UpdateAutoScalingGroupCommand({
      AutoScalingGroupName: group.AutoScalingGroupName,
      MinSize: 0,
    }),
  );
  for (const i of group.Instances) {
    await retry({ intervalInMs: 1000, maxRetries: 30 }, () =>
      autoScalingClient.send(
        new TerminateInstanceInAutoScalingGroupCommand({
          InstanceId: i.InstanceId,
          ShouldDecrementDesiredCapacity: true,
        }),
      ),
    );
  }
}

async function findAutoScalingGroup(groupName) {
  const client = new AutoScalingClient({});
  const paginatedGroups = paginateDescribeAutoScalingGroups({ client }, {});
  for await (const page of paginatedGroups) {
    const group = page.AutoScalingGroups.find(
      (g) => g.AutoScalingGroupName === groupName,
    );
    if (group) {
      return group;
    }
  }
  throw new Error(`Auto scaling group ${groupName} not found.`);
}
```

- For API details, see the following topics in *AWS SDK for JavaScript API Reference*.
 - [AttachLoadBalancerTargetGroups](#)
 - [CreateAutoScalingGroup](#)
 - [CreateInstanceProfile](#)
 - [CreateLaunchTemplate](#)

- [CreateListener](#)
- [CreateLoadBalancer](#)
- [CreateTargetGroup](#)
- [DeleteAutoScalingGroup](#)
- [DeleteInstanceProfile](#)
- [DeleteLaunchTemplate](#)
- [DeleteLoadBalancer](#)
- [DeleteTargetGroup](#)
- [DescribeAutoScalingGroups](#)
- [DescribeAvailabilityZones](#)
- [DescribeIamInstanceProfileAssociations](#)
- [DescribeInstances](#)
- [DescribeLoadBalancers](#)
- [DescribeSubnets](#)
- [DescribeTargetGroups](#)
- [DescribeTargetHealth](#)
- [DescribeVpcs](#)
- [RebootInstances](#)
- [ReplacesIamInstanceProfileAssociation](#)
- [TerminateInstanceInAutoScalingGroup](#)
- [UpdateAutoScalingGroup](#)

Create a user and assume a role

The following code example shows how to create a user and assume a role.

Warning

To avoid security risks, don't use IAM users for authentication when developing purpose-built software or working with real data. Instead, use federation with an identity provider such as [AWS IAM Identity Center](#).

- Create a user with no permissions.
- Create a role that grants permission to list Amazon S3 buckets for the account.
- Add a policy to let the user assume the role.
- Assume the role and list S3 buckets using temporary credentials, then clean up resources.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create an IAM user and a role that grants permission to list Amazon S3 buckets. The user has rights only to assume the role. After assuming the role, use temporary credentials to list buckets for the account.

```
import {
  CreateUserCommand,
  GetUserCommand,
  CreateAccessKeyCommand,
  CreatePolicyCommand,
  CreateRoleCommand,
  AttachRolePolicyCommand,
  DeleteAccessKeyCommand,
  DeleteUserCommand,
  DeleteRoleCommand,
  DeletePolicyCommand,
  DetachRolePolicyCommand,
  IAMClient,
} from "@aws-sdk/client-iam";
import { ListBucketsCommand, S3Client } from "@aws-sdk/client-s3";
import { AssumeRoleCommand, STSClient } from "@aws-sdk/client-sts";
import { retry } from "@aws-doc-sdk-examples/lib/utils/util-timers.js";
import { ScenarioInput } from "@aws-doc-sdk-examples/lib/scenario/index.js";

// Set the parameters.
const iamClient = new IAMClient({});
const userName = "test_name";
const policyName = "test_policy";
```



```
const roleName = "test_role";

/**
 * Create a new IAM user. If the user already exists, give
 * the option to delete and re-create it.
 * @param {string} name
 */
export const createUser = async (name, confirmAll = false) => {
  try {
    const { User } = await iamClient.send(
      new GetUserCommand({ UserName: name }),
    );
    const input = new ScenarioInput(
      "deleteUser",
      "Do you want to delete and remake this user?",
      { type: "confirm" },
    );
    const deleteUser = await input.handle({}, { confirmAll });
    // If the user exists, and you want to delete it, delete the user
    // and then create it again.
    if (deleteUser) {
      await iamClient.send(new DeleteUserCommand({ UserName: User.UserName }));
      await iamClient.send(new CreateUserCommand({ UserName: name }));
    } else {
      console.warn(
        `${name} already exists. The scenario may not work as expected.`
      );
      return User;
    }
  } catch (caught) {
    // If there is no user by that name, create one.
    if (caught instanceof Error && caught.name === "NoSuchEntityException") {
      const { User } = await iamClient.send(
        new CreateUserCommand({ UserName: name }),
      );
      return User;
    } else {
      throw caught;
    }
  }
};

export const main = async (confirmAll = false) => {
  // Create a user. The user has no permissions by default.

```

```
const User = await createUser(userName, confirmAll);

if (!User) {
  throw new Error("User not created");
}

// Create an access key. This key is used to authenticate the new user to
// Amazon Simple Storage Service (Amazon S3) and AWS Security Token Service (AWS
// STS).
// It's not best practice to use access keys. For more information, see https://aws.amazon.com/iam/resources/best-practices/.
const createAccessKeyResponse = await iamClient.send(
  new CreateAccessKeyCommand({ UserName: userName }),
);

if (
  !createAccessKeyResponse.AccessKey?.AccessKeyId ||
  !createAccessKeyResponse.AccessKey?.SecretAccessKey
) {
  throw new Error("Access key not created");
}

const {
  AccessKey: { AccessKeyId, SecretAccessKey },
} = createAccessKeyResponse;

let s3Client = new S3Client({
  credentials: {
    accessKeyId: AccessKeyId,
    secretAccessKey: SecretAccessKey,
  },
});

// Retry the list buckets operation until it succeeds. InvalidAccessKeyId is
// thrown while the user and access keys are still stabilizing.
await retry({ intervalInMs: 1000, maxRetries: 300 }, async () => {
  try {
    return await listBuckets(s3Client);
  } catch (err) {
    if (err instanceof Error && err.name === "InvalidAccessKeyId") {
      throw err;
    }
  }
});
```

```
// Retry the create role operation until it succeeds. A MalformedPolicyDocument
error
// is thrown while the user and access keys are still stabilizing.
const { Role } = await retry(
  {
    intervalInMs: 2000,
    maxRetries: 60,
  },
  () =>
    iamClient.send(
      new CreateRoleCommand({
        AssumeRolePolicyDocument: JSON.stringify({
          Version: "2012-10-17",
          Statement: [
            {
              Effect: "Allow",
              Principal: {
                // Allow the previously created user to assume this role.
                AWS: User.Arn,
              },
              Action: "sts:AssumeRole",
            },
          ],
        }),
        RoleName: roleName,
      }),
    ),
);

if (!Role) {
  throw new Error("Role not created");
}

// Create a policy that allows the user to list S3 buckets.
const { Policy: listBucketPolicy } = await iamClient.send(
  new CreatePolicyCommand({
    PolicyDocument: JSON.stringify({
      Version: "2012-10-17",
      Statement: [
        {
          Effect: "Allow",
          Action: ["s3:ListAllMyBuckets"],
          Resource: "*",
        },
      ],
    }),
  })
);
```

```
    },
  ],
}),
  PolicyName: policyName,
}),
);

if (!listBucketPolicy) {
  throw new Error("Policy not created");
}

// Attach the policy granting the 's3:ListAllMyBuckets' action to the role.
await iamClient.send(
  new AttachRolePolicyCommand({
    PolicyArn: listBucketPolicy.Arn,
    RoleName: Role.RoleName,
  }),
);

// Assume the role.
const stsClient = new STSClient({
  credentials: {
    accessKeyId: AccessKeyId,
    secretAccessKey: SecretAccessKey,
  },
});

// Retry the assume role operation until it succeeds.
const { Credentials } = await retry(
  { intervalInMs: 2000, maxRetries: 60 },
  () =>
    stsClient.send(
      new AssumeRoleCommand({
        RoleArn: Role.Arn,
        RoleSessionName: `iamBasicScenarioSession-${Math.floor(
          Math.random() * 1000000,
        )}`,
        DurationSeconds: 900,
      }),
    ),
);

if (!Credentials?.AccessKeyId || !Credentials?.SecretAccessKey) {
  throw new Error("Credentials not created");
}
```

```
}

s3Client = new S3Client({
  credentials: {
    accessKeyId: Credentials.AccessKeyId,
    secretAccessKey: Credentials.SecretAccessKey,
    sessionToken: Credentials.SessionToken,
  },
});

// List the S3 buckets again.
// Retry the list buckets operation until it succeeds. AccessDenied might
// be thrown while the role policy is still stabilizing.
await retry({ intervalInMs: 2000, maxRetries: 60 }, () =>
  listBuckets(s3Client),
);

// Clean up.
await iamClient.send(
  new DetachRolePolicyCommand({
    PolicyArn: listBucketPolicy.Arn,
    RoleName: Role.RoleName,
  }),
);

await iamClient.send(
  new DeletePolicyCommand({
    PolicyArn: listBucketPolicy.Arn,
  }),
);

await iamClient.send(
  new DeleteRoleCommand({
    RoleName: Role.RoleName,
  }),
);

await iamClient.send(
  new DeleteAccessKeyCommand({
    UserName: userName,
    AccessKeyId,
  }),
);
```

```
    await iamClient.send(
      new DeleteUserCommand({
        UserName: userName,
      }),
    );
  };

  /**
   *
   * @param {S3Client} s3Client
   */
  const listBuckets = async (s3Client) => {
    const { Buckets } = await s3Client.send(new ListBucketsCommand({}));

    if (!Buckets) {
      throw new Error("Buckets not listed");
    }

    console.log(Buckets.map((bucket) => bucket.Name).join("\n"));
  };
};
```

- For API details, see the following topics in *AWS SDK for JavaScript API Reference*.
 - [AttachRolePolicy](#)
 - [CreateAccessKey](#)
 - [CreatePolicy](#)
 - [CreateRole](#)
 - [CreateUser](#)
 - [DeleteAccessKey](#)
 - [DeletePolicy](#)
 - [DeleteRole](#)
 - [DeleteUser](#)
 - [DeleteUserPolicy](#)
 - [DetachRolePolicy](#)
 - [PutUserPolicy](#)

Kinesis examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with Kinesis.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Topics

- [Serverless examples](#)

Serverless examples

Invoke a Lambda function from a Kinesis trigger

The following code example shows how to implement a Lambda function that receives an event triggered by receiving records from a Kinesis stream. The function retrieves the Kinesis payload, decodes from Base64, and logs the record contents.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [Serverless examples](#) repository.

Consuming a Kinesis event with Lambda using JavaScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    try {
      console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      console.log(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      console.error(`An error occurred ${err}`);
    }
  }
}
```

```

        throw err;
    }
}
console.log(`Successfully processed ${event.Records.length} records.`);
};

async function getRecordDataAsync(payload) {
    var data = Buffer.from(payload.data, "base64").toString("utf-8");
    await Promise.resolve(1); //Placeholder for actual async work
    return data;
}

```

Consuming a Kinesis event with Lambda using TypeScript.

```

// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import {
    KinesisStreamEvent,
    Context,
    KinesisStreamHandler,
    KinesisStreamRecordPayload,
} from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
    logLevel: "INFO",
    serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (
    event: KinesisStreamEvent,
    context: Context
): Promise<void> => {
    for (const record of event.Records) {
        try {
            logger.info(`Processed Kinesis Event - EventID: ${record.eventID}`);
            const recordData = await getRecordDataAsync(record.kinesis);
            logger.info(`Record Data: ${recordData}`);
            // TODO: Do interesting work based on the new data
        } catch (err) {
            logger.error(`An error occurred ${err}`);
        }
    }
}

```



```
        throw err;
    }
    logger.info(`Successfully processed ${event.Records.length} records.`);
}
};

async function getRecordDataAsync(
    payload: KinesisStreamRecordPayload
): Promise<string> {
    var data = Buffer.from(payload.data, "base64").toString("utf-8");
    await Promise.resolve(1); //Placeholder for actual async work
    return data;
}
```

Reporting batch item failures for Lambda functions with a Kinesis trigger

The following code example shows how to implement partial batch response for Lambda functions that receive events from a Kinesis stream. The function reports the batch item failures in the response, signaling to Lambda to retry those messages later.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [Serverless examples](#) repository.

Reporting Kinesis batch item failures with Lambda using Javascript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
    for (const record of event.Records) {
        try {
            console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
            const recordData = await getRecordDataAsync(record.kinesis);
            console.log(`Record Data: ${recordData}`);
            // TODO: Do interesting work based on the new data
        } catch (err) {
```

```

        console.error(`An error occurred ${err}`);
        /* Since we are working with streams, we can return the failed item
        immediately.
           Lambda will immediately begin to retry processing from this failed item
        onwards. */
        return {
            batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
        };
    }
}
console.log(`Successfully processed ${event.Records.length} records.`);
return { batchItemFailures: [] };
};

async function getRecordDataAsync(payload) {
    var data = Buffer.from(payload.data, "base64").toString("utf-8");
    await Promise.resolve(1); //Placeholder for actual async work
    return data;
}

```

Reporting Kinesis batch item failures with Lambda using TypeScript.

```

// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import {
    KinesisStreamEvent,
    Context,
    KinesisStreamHandler,
    KinesisStreamRecordPayload,
    KinesisStreamBatchResponse,
} from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
    logLevel: "INFO",
    serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (
    event: KinesisStreamEvent,
    context: Context

```

```
) : Promise<KinesisStreamBatchResponse> => {
  for (const record of event.Records) {
    try {
      logger.info(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      logger.info(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      logger.error(`An error occurred ${err}`);
      /* Since we are working with streams, we can return the failed item
      immediately.
      Lambda will immediately begin to retry processing from this failed item
      onwards. */
      return {
        batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
      };
    }
  }
  logger.info(`Successfully processed ${event.Records.length} records.`);
  return { batchItemFailures: [] };
};

async function getRecordDataAsync(
  payload: KinesisStreamRecordPayload
): Promise<string> {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

Lambda examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with Lambda.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Scenarios are code examples that show you how to accomplish specific tasks by calling multiple functions within a service or combined with other AWS services.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Get started

Hello Lambda

The following code examples show how to get started using Lambda.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { LambdaClient, paginateListFunctions } from "@aws-sdk/client-lambda";

const client = new LambdaClient({});

export const helloLambda = async () => {
  const paginator = paginateListFunctions({ client }, {});
  const functions = [];

  for await (const page of paginator) {
    const funcNames = page.Functions.map((f) => f.FunctionName);
    functions.push(...funcNames);
  }

  console.log("Functions:");
  console.log(functions.join("\n"));
  return functions;
};
```

- For API details, see [ListFunctions](#) in *AWS SDK for JavaScript API Reference*.

Topics

- [Actions](#)

- [Scenarios](#)
- [Serverless examples](#)

Actions

CreateFunction

The following code example shows how to use CreateFunction.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const createFunction = async (funcName, roleArn) => {
  const client = new LambdaClient({});
  const code = await readFile(`${dirname}../functions/${funcName}.zip`);

  const command = new CreateFunctionCommand({
    Code: { ZipFile: code },
    FunctionName: funcName,
    Role: roleArn,
    Architectures: [Architecture.arm64],
    Handler: "index.handler", // Required when sending a .zip file
    PackageType: PackageType.Zip, // Required when sending a .zip file
    Runtime: Runtime.nodejs16x, // Required when sending a .zip file
  });

  return client.send(command);
};
```

- For API details, see [CreateFunction](#) in *AWS SDK for JavaScript API Reference*.

DeleteFunction

The following code example shows how to use DeleteFunction.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
/**
 * @param {string} funcName
 */
const deleteFunction = (funcName) => {
  const client = new LambdaClient({});
  const command = new DeleteFunctionCommand({ FunctionName: funcName });
  return client.send(command);
};
```

- For API details, see [DeleteFunction](#) in *AWS SDK for JavaScript API Reference*.

GetFunction

The following code example shows how to use GetFunction.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const getFunction = (funcName) => {
  const client = new LambdaClient({});
  const command = new GetFunctionCommand({ FunctionName: funcName });
  return client.send(command);
};
```

- For API details, see [GetFunction](#) in *AWS SDK for JavaScript API Reference*.

Invoke

The following code example shows how to use Invoke.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const invoke = async (funcName, payload) => {
  const client = new LambdaClient({});
  const command = new InvokeCommand({
    FunctionName: funcName,
    Payload: JSON.stringify(payload),
    LogType: LogType.Tail,
  });

  const { Payload, LogResult } = await client.send(command);
  const result = Buffer.from(Payload).toString();
  const logs = Buffer.from(LogResult, "base64").toString();
  return { logs, result };
};
```

- For API details, see [Invoke](#) in *AWS SDK for JavaScript API Reference*.

ListFunctions

The following code example shows how to use ListFunctions.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const listFunctions = () => {
  const client = new LambdaClient({});
  const command = new ListFunctionsCommand({});

  return client.send(command);
};
```

- For API details, see [ListFunctions](#) in *AWS SDK for JavaScript API Reference*.

UpdateFunctionCode

The following code example shows how to use UpdateFunctionCode.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const updateFunctionCode = async (funcName, newFunc) => {
  const client = new LambdaClient({});
  const code = await readFile(`${dirname}../functions/${newFunc}.zip`);
  const command = new UpdateFunctionCodeCommand({
    ZipFile: code,
    FunctionName: funcName,
    Architectures: [Architecture.arm64],
    Handler: "index.handler", // Required when sending a .zip file
    PackageType: PackageType.Zip, // Required when sending a .zip file
    Runtime: Runtime.nodejs16x, // Required when sending a .zip file
  });

  return client.send(command);
};
```

- For API details, see [UpdateFunctionCode](#) in *AWS SDK for JavaScript API Reference*.

UpdateFunctionConfiguration

The following code example shows how to use `UpdateFunctionConfiguration`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
const updateFunctionConfiguration = (funcName) => {
  const client = new LambdaClient({});
  const config = readFileSync(`${dirname}../functions/config.json`).toString();
  const command = new UpdateFunctionConfigurationCommand({
    ...JSON.parse(config),
    FunctionName: funcName,
  });
  return client.send(command);
};
```

- For API details, see [UpdateFunctionConfiguration](#) in *AWS SDK for JavaScript API Reference*.

Scenarios

Create a serverless application to manage photos

The following code example shows how to create a serverless application that lets users manage photos using labels.

SDK for JavaScript (v3)

Shows how to develop a photo asset management application that detects labels in images using Amazon Rekognition and stores them for later retrieval.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

For a deep dive into the origin of this example see the post on [AWS Community](#).

Services used in this example

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Create an application to analyze customer feedback

The following code example shows how to create an application that analyzes customer comment cards, translates them from their original language, determines their sentiment, and generates an audio file from the translated text.

SDK for JavaScript (v3)

This example application analyzes and stores customer feedback cards. Specifically, it fulfills the need of a fictitious hotel in New York City. The hotel receives feedback from guests in various languages in the form of physical comment cards. That feedback is uploaded into the app through a web client. After an image of a comment card is uploaded, the following steps occur:

- Text is extracted from the image using Amazon Textract.
- Amazon Comprehend determines the sentiment of the extracted text and its language.
- The extracted text is translated to English using Amazon Translate.
- Amazon Polly synthesizes an audio file from the extracted text.

The full app can be deployed with the AWS CDK. For source code and deployment instructions, see the project in [GitHub](#). The following excerpts show how the AWS SDK for JavaScript is used inside of Lambda functions.

```
import {
  ComprehendClient,
  DetectDominantLanguageCommand,
  DetectSentimentCommand,
} from "@aws-sdk/client-comprehend";

/**
 * Determine the language and sentiment of the extracted text.
```

```

*
* @param {{ source_text: string }} extractTextOutput
*/
export const handler = async (extractTextOutput) => {
  const comprehendClient = new ComprehendClient({});

  const detectDominantLanguageCommand = new DetectDominantLanguageCommand({
    Text: extractTextOutput.source_text,
  });

  // The source language is required for sentiment analysis and
  // translation in the next step.
  const { Languages } = await comprehendClient.send(
    detectDominantLanguageCommand,
  );

  const languageCode = Languages[0].LanguageCode;

  const detectSentimentCommand = new DetectSentimentCommand({
    Text: extractTextOutput.source_text,
    LanguageCode: languageCode,
  });

  const { Sentiment } = await comprehendClient.send(detectSentimentCommand);

  return {
    sentiment: Sentiment,
    language_code: languageCode,
  };
};

```

```

import {
  DetectDocumentTextCommand,
  TextractClient,
} from "@aws-sdk/client-textract";

/**
 * Fetch the S3 object from the event and analyze it using Amazon Textract.
 *
 * @param {import("@types/aws-lambda").EventBridgeEvent<"Object Created">}
  eventBridgeS3Event
 */
export const handler = async (eventBridgeS3Event) => {

```

```

const textractClient = new TextractClient();

const detectDocumentTextCommand = new DetectDocumentTextCommand({
  Document: {
    S3Object: {
      Bucket: eventBridgeS3Event.bucket,
      Name: eventBridgeS3Event.object,
    },
  },
});

// Textract returns a list of blocks. A block can be a line, a page, word, etc.
// Each block also contains geometry of the detected text.
// For more information on the Block type, see https://docs.aws.amazon.com/
textract/latest/dg/API_Block.html.
const { Blocks } = await textractClient.send(detectDocumentTextCommand);

// For the purpose of this example, we are only interested in words.
const extractedWords = Blocks.filter((b) => b.BlockType === "WORD").map(
  (b) => b.Text,
);

return extractedWords.join(" ");
};

```

```

import { PollyClient, SynthesizeSpeechCommand } from "@aws-sdk/client-polly";
import { S3Client } from "@aws-sdk/client-s3";
import { Upload } from "@aws-sdk/lib-storage";

/**
 * Synthesize an audio file from text.
 *
 * @param {{ bucket: string, translated_text: string, object: string }}
  sourceDestinationConfig
 */
export const handler = async (sourceDestinationConfig) => {
  const pollyClient = new PollyClient({});

  const synthesizeSpeechCommand = new SynthesizeSpeechCommand({
    Engine: "neural",
    Text: sourceDestinationConfig.translated_text,
    VoiceId: "Ruth",
    OutputFormat: "mp3",
  });

```

```
});

const { AudioStream } = await pollyClient.send(synthesizeSpeechCommand);

const audioKey = `${sourceDestinationConfig.object}.mp3`;

// Store the audio file in S3.
const s3Client = new S3Client();
const upload = new Upload({
  client: s3Client,
  params: {
    Bucket: sourceDestinationConfig.bucket,
    Key: audioKey,
    Body: AudioStream,
    ContentType: "audio/mp3",
  },
});

await upload.done();
return audioKey;
};
```

```
import {
  TranslateClient,
  TranslateTextCommand,
} from "@aws-sdk/client-translate";

/**
 * Translate the extracted text to English.
 *
 * @param {{ extracted_text: string, source_language_code: string }}
  textAndSourceLanguage
 */
export const handler = async (textAndSourceLanguage) => {
  const translateClient = new TranslateClient({});

  const translateCommand = new TranslateTextCommand({
    SourceLanguageCode: textAndSourceLanguage.source_language_code,
    TargetLanguageCode: "en",
    Text: textAndSourceLanguage.extracted_text,
  });

  const { TranslatedText } = await translateClient.send(translateCommand);
```

```
    return { translated_text: TranslatedText };  
};
```

Services used in this example

- Amazon Comprehend
- Lambda
- Amazon Polly
- Amazon Textract
- Amazon Translate

Get started with functions

The following code example shows how to:

- Create an IAM role and Lambda function, then upload handler code.
- Invoke the function with a single parameter and get results.
- Update the function code and configure with an environment variable.
- Invoke the function with new parameters and get results. Display the returned execution log.
- List the functions for your account, then clean up resources.

For more information, see [Create a Lambda function with the console](#).

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create an AWS Identity and Access Management (IAM) role that grants Lambda permission to write to logs.

```
log(`Creating role (${NAME_ROLE_LAMBDA})...`);  
const response = await createRole(NAME_ROLE_LAMBDA);
```

```
import { AttachRolePolicyCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
 * @param {string} policyArn
 * @param {string} roleName
 */
export const attachRolePolicy = (policyArn, roleName) => {
  const command = new AttachRolePolicyCommand({
    PolicyArn: policyArn,
    RoleName: roleName,
  });

  return client.send(command);
};
```

Create a Lambda function and upload handler code.

```
const createFunction = async (funcName, roleArn) => {
  const client = new LambdaClient({});
  const code = await readFile(`${dirname}../functions/${funcName}.zip`);

  const command = new CreateFunctionCommand({
    Code: { ZipFile: code },
    FunctionName: funcName,
    Role: roleArn,
    Architectures: [Architecture.arm64],
    Handler: "index.handler", // Required when sending a .zip file
    PackageType: PackageType.Zip, // Required when sending a .zip file
    Runtime: Runtime.nodejs16x, // Required when sending a .zip file
  });

  return client.send(command);
};
```

Invoke the function with a single parameter and get results.

```
const invoke = async (funcName, payload) => {
```

```
const client = new LambdaClient({});
const command = new InvokeCommand({
  FunctionName: funcName,
  Payload: JSON.stringify(payload),
  LogType: LogType.Tail,
});

const { Payload, LogResult } = await client.send(command);
const result = Buffer.from(Payload).toString();
const logs = Buffer.from(LogResult, "base64").toString();
return { logs, result };
};
```

Update the function code and configure its Lambda environment with an environment variable.

```
const updateFunctionCode = async (funcName, newFunc) => {
  const client = new LambdaClient({});
  const code = await readFile(`${dirname}../functions/${newFunc}.zip`);
  const command = new UpdateFunctionCodeCommand({
    ZipFile: code,
    FunctionName: funcName,
    Architectures: [Architecture.arm64],
    Handler: "index.handler", // Required when sending a .zip file
    PackageType: PackageType.Zip, // Required when sending a .zip file
    Runtime: Runtime.nodejs16x, // Required when sending a .zip file
  });

  return client.send(command);
};

const updateFunctionConfiguration = (funcName) => {
  const client = new LambdaClient({});
  const config = readFileSync(`${dirname}../functions/config.json`).toString();
  const command = new UpdateFunctionConfigurationCommand({
    ...JSON.parse(config),
    FunctionName: funcName,
  });
  return client.send(command);
};
```

List the functions for your account.


```
const listFunctions = () => {
  const client = new LambdaClient({});
  const command = new ListFunctionsCommand({});

  return client.send(command);
};
```

Delete the IAM role and the Lambda function.

```
import { DeleteRoleCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
 * @param {string} roleName
 */
export const deleteRole = (roleName) => {
  const command = new DeleteRoleCommand({ RoleName: roleName });
  return client.send(command);
};

/**
 * @param {string} funcName
 */
const deleteFunction = (funcName) => {
  const client = new LambdaClient({});
  const command = new DeleteFunctionCommand({ FunctionName: funcName });
  return client.send(command);
};
```

• For API details, see the following topics in *AWS SDK for JavaScript API Reference*.

- [CreateFunction](#)
- [DeleteFunction](#)
- [GetFunction](#)
- [Invoke](#)
- [ListFunctions](#)
- [UpdateFunctionCode](#)

- [UpdateFunctionConfiguration](#)

Invoke a Lambda function from a browser

The following code example shows how to invoke an AWS Lambda function from a browser.

SDK for JavaScript (v2)

You can create a browser-based application that uses an AWS Lambda function to update an Amazon DynamoDB table with user selections.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- DynamoDB
- Lambda

SDK for JavaScript (v3)

You can create a browser-based application that uses an AWS Lambda function to update an Amazon DynamoDB table with user selections. This app uses AWS SDK for JavaScript v3.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- DynamoDB
- Lambda

Use API Gateway to invoke a Lambda function

The following code example shows how to create an AWS Lambda function invoked by Amazon API Gateway.

SDK for JavaScript (v3)

Shows how to create an AWS Lambda function by using the Lambda JavaScript runtime API. This example invokes different AWS services to perform a specific use case. This example

demonstrates how to create a Lambda function invoked by Amazon API Gateway that scans an Amazon DynamoDB table for work anniversaries and uses Amazon Simple Notification Service (Amazon SNS) to send a text message to your employees that congratulates them at their one year anniversary date.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

This example is also available in the [AWS SDK for JavaScript v3 developer guide](#).

Services used in this example

- API Gateway
- DynamoDB
- Lambda
- Amazon SNS

Use scheduled events to invoke a Lambda function

The following code example shows how to create an AWS Lambda function invoked by an Amazon EventBridge scheduled event.

SDK for JavaScript (v3)

Shows how to create an Amazon EventBridge scheduled event that invokes an AWS Lambda function. Configure EventBridge to use a cron expression to schedule when the Lambda function is invoked. In this example, you create a Lambda function by using the Lambda JavaScript runtime API. This example invokes different AWS services to perform a specific use case. This example demonstrates how to create an app that sends a mobile text message to your employees that congratulates them at the one year anniversary date.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

This example is also available in the [AWS SDK for JavaScript v3 developer guide](#).

Services used in this example

- DynamoDB
- EventBridge

- Lambda
- Amazon SNS

Serverless examples

Connecting to an Amazon RDS database in a Lambda function

The following code example shows how to implement a Lambda function that connects to an RDS database. The function makes a simple database request and returns the result.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [Serverless examples](#) repository.

Connecting to an Amazon RDS database in a Lambda function using JavaScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
/*
Node.js code here.
*/
// ES6+ example
import { Signer } from "@aws-sdk/rds-signer";
import mysql from 'mysql2/promise';

async function createAuthToken() {
  // Define connection authentication parameters
  const dbinfo = {

    hostname: process.env.ProxyHostName,
    port: process.env.Port,
    username: process.env.DBUserName,
    region: process.env.AWS_REGION,

  }

  // Create RDS Signer object
```

```
const signer = new Signer(dbinfo);

// Request authorization token from RDS, specifying the username
const token = await signer.getAuthToken();
return token;
}

async function dbOps() {

  // Obtain auth token
  const token = await createAuthToken();
  // Define connection configuration
  let connectionConfig = {
    host: process.env.ProxyHostName,
    user: process.env.DBUserName,
    password: token,
    database: process.env.DBName,
    ssl: 'Amazon RDS'
  }
  // Create the connection to the DB
  const conn = await mysql.createConnection(connectionConfig);
  // Obtain the result of the query
  const [res,] = await conn.execute('select ?+? as sum', [3, 2]);
  return res;
}

export const handler = async (event) => {
  // Execute database flow
  const result = await dbOps();
  // Return result
  return {
    statusCode: 200,
    body: JSON.stringify("The selected sum is: " + result[0].sum)
  }
};
```

Connecting to an Amazon RDS database in a Lambda function using TypeScript.

```
import { Signer } from "@aws-sdk/rds-signer";
import mysql from 'mysql2/promise';
```

```
// RDS settings
// Using '!' (non-null assertion operator) to tell the TypeScript compiler that the
// DB settings are not null or undefined,
const proxy_host_name = process.env.PROXY_HOST_NAME!
const port = parseInt(process.env.PORT!)
const db_name = process.env.DB_NAME!
const db_user_name = process.env.DB_USER_NAME!
const aws_region = process.env.AWS_REGION!

async function createAuthToken(): Promise<string> {

  // Create RDS Signer object
  const signer = new Signer({
    hostname: proxy_host_name,
    port: port,
    region: aws_region,
    username: db_user_name
  });

  // Request authorization token from RDS, specifying the username
  const token = await signer.getAuthToken();
  return token;
}

async function dbOps(): Promise<mysql.QueryResult | undefined> {
  try {
    // Obtain auth token
    const token = await createAuthToken();
    const conn = await mysql.createConnection({
      host: proxy_host_name,
      user: db_user_name,
      password: token,
      database: db_name,
      ssl: 'Amazon RDS' // Ensure you have the CA bundle for SSL connection
    });
    const [rows, fields] = await conn.execute('SELECT ? + ? AS sum', [3, 2]);
    console.log('result:', rows);
    return rows;
  }
  catch (err) {
    console.log(err);
  }
}
```

```
}

export const lambdaHandler = async (event: any): Promise<{ statusCode: number; body:
string }> => {
  // Execute database flow
  const result = await dbOps();

  // Return error if result is undefined
  if (result == undefined)
    return {
      statusCode: 500,
      body: JSON.stringify(`Error with connection to DB host`)
    }

  // Return result
  return {
    statusCode: 200,
    body: JSON.stringify(`The selected sum is: ${result[0].sum}`)
  };
};
```

Invoke a Lambda function from a Kinesis trigger

The following code example shows how to implement a Lambda function that receives an event triggered by receiving records from a Kinesis stream. The function retrieves the Kinesis payload, decodes from Base64, and logs the record contents.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [Serverless examples](#) repository.

Consuming a Kinesis event with Lambda using JavaScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
```

```

    try {
      console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      console.log(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      console.error(`An error occurred ${err}`);
      throw err;
    }
  }
  console.log(`Successfully processed ${event.Records.length} records.`);
};

async function getRecordDataAsync(payload) {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}

```

Consuming a Kinesis event with Lambda using TypeScript.

```

// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import {
  KinesisStreamEvent,
  Context,
  KinesisStreamHandler,
  KinesisStreamRecordPayload,
} from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
  serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (
  event: KinesisStreamEvent,
  context: Context
): Promise<void> => {
  for (const record of event.Records) {

```



```
    try {
      logger.info(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      logger.info(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      logger.error(`An error occurred ${err}`);
      throw err;
    }
    logger.info(`Successfully processed ${event.Records.length} records.`);
  }
};

async function getRecordDataAsync(
  payload: KinesisStreamRecordPayload
): Promise<string> {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

Invoke a Lambda function from a DynamoDB trigger

The following code example shows how to implement a Lambda function that receives an event triggered by receiving records from a DynamoDB stream. The function retrieves the DynamoDB payload and logs the record contents.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [Serverless examples](#) repository.

Consuming a DynamoDB event with Lambda using JavaScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
```

```
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(record => {
      logDynamoDBRecord(record);
    });
  };

const logDynamoDBRecord = (record) => {
  console.log(record.eventID);
  console.log(record.eventName);
  console.log(`DynamoDB Record: ${JSON.stringify(record.dynamodb)}`);
};
```

Consuming a DynamoDB event with Lambda using TypeScript.

```
export const handler = async (event, context) => {
  console.log(JSON.stringify(event, null, 2));
  event.Records.forEach(record => {
    logDynamoDBRecord(record);
  });
}

const logDynamoDBRecord = (record) => {
  console.log(record.eventID);
  console.log(record.eventName);
  console.log(`DynamoDB Record: ${JSON.stringify(record.dynamodb)}`);
};
```

Invoke a Lambda function from a Amazon DocumentDB trigger

The following code example shows how to implement a Lambda function that receives an event triggered by receiving records from a DocumentDB change stream. The function retrieves the DocumentDB payload and logs the record contents.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [Serverless examples](#) repository.

Consuming a Amazon DocumentDB event with Lambda using JavaScript.

```
console.log('Loading function');
exports.handler = async (event, context) => {
  event.events.forEach(record => {
    logDocumentDBEvent(record);
  });
  return 'OK';
};

const logDocumentDBEvent = (record) => {
  console.log('Operation type: ' + record.event.operationType);
  console.log('db: ' + record.event.ns.db);
  console.log('collection: ' + record.event.ns.coll);
  console.log('Full document:', JSON.stringify(record.event.fullDocument, null,
  2));
};
```

Consuming a Amazon DocumentDB event with Lambda using TypeScript

```
import { DocumentDBEventRecord, DocumentDBEventSubscriptionContext } from 'aws-
lambda';

console.log('Loading function');

export const handler = async (
  event: DocumentDBEventSubscriptionContext,
  context: any
): Promise<string> => {
  event.events.forEach((record: DocumentDBEventRecord) => {
    logDocumentDBEvent(record);
  });
  return 'OK';
};

const logDocumentDBEvent = (record: DocumentDBEventRecord): void => {
  console.log('Operation type: ' + record.event.operationType);
  console.log('db: ' + record.event.ns.db);
  console.log('collection: ' + record.event.ns.coll);
  console.log('Full document:', JSON.stringify(record.event.fullDocument, null, 2));
};
```

Invoke a Lambda function from an Amazon MSK trigger

The following code example shows how to implement a Lambda function that receives an event triggered by receiving records from an Amazon MSK cluster. The function retrieves the MSK payload and logs the record contents.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [Serverless examples](#) repository.

Consuming an Amazon MSK event with Lambda using JavaScript.

```
exports.handler = async (event) => {
  // Iterate through keys
  for (let key in event.records) {
    console.log('Key: ', key)
    // Iterate through records
    event.records[key].map((record) => {
      console.log('Record: ', record)
      // Decode base64
      const msg = Buffer.from(record.value, 'base64').toString()
      console.log('Message:', msg)
    })
  }
}
```

Invoke a Lambda function from an Amazon S3 trigger

The following code example shows how to implement a Lambda function that receives an event triggered by uploading an object to an S3 bucket. The function retrieves the S3 bucket name and object key from the event parameter and calls the Amazon S3 API to retrieve and log the content type of the object.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [Serverless examples](#) repository.

Consuming an S3 event with Lambda using JavaScript.

```
import { S3Client, HeadObjectCommand } from "@aws-sdk/client-s3";

const client = new S3Client();

export const handler = async (event, context) => {

  // Get the object from the event and show its content type
  const bucket = event.Records[0].s3.bucket.name;
  const key = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/g, ' '));

  try {
    const { ContentType } = await client.send(new HeadObjectCommand({
      Bucket: bucket,
      Key: key,
    }));

    console.log('CONTENT TYPE:', ContentType);
    return ContentType;

  } catch (err) {
    console.log(err);
    const message = `Error getting object ${key} from bucket ${bucket}. Make sure they exist and your bucket is in the same region as this function.`;
    console.log(message);
    throw new Error(message);
  }
};
```

Consuming an S3 event with Lambda using TypeScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { S3Event } from 'aws-lambda';
import { S3Client, HeadObjectCommand } from '@aws-sdk/client-s3';

const s3 = new S3Client({ region: process.env.AWS_REGION });

export const handler = async (event: S3Event): Promise<string | undefined> => {
  // Get the object from the event and show its content type
  const bucket = event.Records[0].s3.bucket.name;
  const key = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/g, '
  '));
  const params = {
    Bucket: bucket,
    Key: key,
  };
  try {
    const { ContentType } = await s3.send(new HeadObjectCommand(params));
    console.log('CONTENT TYPE:', ContentType);
    return ContentType;
  } catch (err) {
    console.log(err);
    const message = `Error getting object ${key} from bucket ${bucket}. Make sure
they exist and your bucket is in the same region as this function.`;
    console.log(message);
    throw new Error(message);
  }
};
```

Invoke a Lambda function from an Amazon SNS trigger

The following code example shows how to implement a Lambda function that receives an event triggered by receiving messages from an SNS topic. The function retrieves the messages from the event parameter and logs the content of each message.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [Serverless examples](#) repository.

Consuming an SNS event with Lambda using JavaScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    await processMessageAsync(record);
  }
  console.info("done");
};

async function processMessageAsync(record) {
  try {
    const message = JSON.stringify(record.Sns.Message);
    console.log(`Processed message ${message}`);
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

Consuming an SNS event with Lambda using TypeScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SNSEvent, Context, SNSHandler, SNSEventRecord } from "aws-lambda";

export const functionHandler: SNSHandler = async (
  event: SNSEvent,
  context: Context
): Promise<void> => {
  for (const record of event.Records) {
```

```
    await processMessageAsync(record);
  }
  console.info("done");
};

async function processMessageAsync(record: SNSEventRecord): Promise<any> {
  try {
    const message: string = JSON.stringify(record.Sns.Message);
    console.log(`Processed message ${message}`);
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

Invoke a Lambda function from an Amazon SQS trigger

The following code example shows how to implement a Lambda function that receives an event triggered by receiving messages from an SQS queue. The function retrieves the messages from the event parameter and logs the content of each message.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [Serverless examples](#) repository.

Consuming an SQS event with Lambda using JavaScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const message of event.Records) {
    await processMessageAsync(message);
  }
  console.info("done");
};
```



```
async function processMessageAsync(message) {
  try {
    console.log(`Processed message ${message.body}`);
    // TODO: Do interesting work based on the new message
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

Consuming an SQS event with Lambda using TypeScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SQSEvent, Context, SQSHandler, SQSRecord } from "aws-lambda";

export const functionHandler: SQSHandler = async (
  event: SQSEvent,
  context: Context
): Promise<void> => {
  for (const message of event.Records) {
    await processMessageAsync(message);
  }
  console.info("done");
};

async function processMessageAsync(message: SQSRecord): Promise<any> {
  try {
    console.log(`Processed message ${message.body}`);
    // TODO: Do interesting work based on the new message
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

Reporting batch item failures for Lambda functions with a Kinesis trigger

The following code example shows how to implement partial batch response for Lambda functions that receive events from a Kinesis stream. The function reports the batch item failures in the response, signaling to Lambda to retry those messages later.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [Serverless examples](#) repository.

Reporting Kinesis batch item failures with Lambda using Javascript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    try {
      console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      console.log(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      console.error(`An error occurred ${err}`);
      /* Since we are working with streams, we can return the failed item
      immediately.
      Lambda will immediately begin to retry processing from this failed item
      onwards. */
      return {
        batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
      };
    }
  }
  console.log(`Successfully processed ${event.Records.length} records.`);
  return { batchItemFailures: [] };
};

async function getRecordDataAsync(payload) {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
}
```

```
    await Promise.resolve(1); //Placeholder for actual async work
    return data;
}
```

Reporting Kinesis batch item failures with Lambda using TypeScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import {
    KinesisStreamEvent,
    Context,
    KinesisStreamHandler,
    KinesisStreamRecordPayload,
    KinesisStreamBatchResponse,
} from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
    logLevel: "INFO",
    serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (
    event: KinesisStreamEvent,
    context: Context
): Promise<KinesisStreamBatchResponse> => {
    for (const record of event.Records) {
        try {
            logger.info(`Processed Kinesis Event - EventID: ${record.eventID}`);
            const recordData = await getRecordDataAsync(record.kinesis);
            logger.info(`Record Data: ${recordData}`);
            // TODO: Do interesting work based on the new data
        } catch (err) {
            logger.error(`An error occurred ${err}`);
            /* Since we are working with streams, we can return the failed item
            immediately.
            Lambda will immediately begin to retry processing from this failed item
            onwards. */
            return {
                batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
            };
        }
    }
}
```

```
    }  
  }  
  logger.info(`Successfully processed ${event.Records.length} records.`);  
  return { batchItemFailures: [] };  
};  
  
async function getRecordDataAsync(  
  payload: KinesisStreamRecordPayload  
) : Promise<string> {  
  var data = Buffer.from(payload.data, "base64").toString("utf-8");  
  await Promise.resolve(1); //Placeholder for actual async work  
  return data;  
}
```

Reporting batch item failures for Lambda functions with a DynamoDB trigger

The following code example shows how to implement partial batch response for Lambda functions that receive events from a DynamoDB stream. The function reports the batch item failures in the response, signaling to Lambda to retry those messages later.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [Serverless examples](#) repository.

Reporting DynamoDB batch item failures with Lambda using JavaScript.

```
export const handler = async (event) => {  
  const records = event.Records;  
  let curRecordSequenceNumber = "";  
  
  for (const record of records) {  
    try {  
      // Process your record  
      curRecordSequenceNumber = record.dynamodb.SequenceNumber;  
    } catch (e) {  
      // Return failed record's sequence number
```

```
        return { batchItemFailures: [{ itemIdentifier: curRecordSequenceNumber }] };
    }
}

return { batchItemFailures: [] };
};
```

Reporting DynamoDB batch item failures with Lambda using TypeScript.

```
import {
    DynamoDBBatchResponse,
    DynamoDBBatchItemFailure,
    DynamoDBStreamEvent,
} from "aws-lambda";

export const handler = async (
    event: DynamoDBStreamEvent
): Promise<DynamoDBBatchResponse> => {
    const batchItemFailures: DynamoDBBatchItemFailure[] = [];
    let curRecordSequenceNumber;

    for (const record of event.Records) {
        curRecordSequenceNumber = record.dynamodb?.SequenceNumber;

        if (curRecordSequenceNumber) {
            batchItemFailures.push({
                itemIdentifier: curRecordSequenceNumber,
            });
        }
    }

    return { batchItemFailures: batchItemFailures };
};
```

Reporting batch item failures for Lambda functions with an Amazon SQS trigger

The following code example shows how to implement partial batch response for Lambda functions that receive events from an SQS queue. The function reports the batch item failures in the response, signaling to Lambda to retry those messages later.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [Serverless examples](#) repository.

Reporting SQS batch item failures with Lambda using JavaScript.

```
// Node.js 20.x Lambda runtime, AWS SDK for Javascript V3
export const handler = async (event, context) => {
  const batchItemFailures = [];
  for (const record of event.Records) {
    try {
      await processMessageAsync(record, context);
    } catch (error) {
      batchItemFailures.push({ itemIdentifier: record.messageId });
    }
  }
  return { batchItemFailures };
};

async function processMessageAsync(record, context) {
  if (record.body && record.body.includes("error")) {
    throw new Error("There is an error in the SQS Message.");
  }
  console.log(`Processed message: ${record.body}`);
}
```

Reporting SQS batch item failures with Lambda using TypeScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SQSEvent, SQSBatchResponse, Context, SQSBatchItemFailure, SQSRecord } from
  'aws-lambda';

export const handler = async (event: SQSEvent, context: Context):
  Promise<SQSBatchResponse> => {
  const batchItemFailures: SQSBatchItemFailure[] = [];

  for (const record of event.Records) {
```

```
        try {
            await processMessageAsync(record);
        } catch (error) {
            batchItemFailures.push({ itemIdentifier: record.messageId });
        }
    }

    return {batchItemFailures: batchItemFailures};
};

async function processMessageAsync(record: SQSRecord): Promise<void> {
    if (record.body && record.body.includes("error")) {
        throw new Error('There is an error in the SQS Message.');
```

Amazon Lex examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with Amazon Lex.

Scenarios are code examples that show you how to accomplish specific tasks by calling multiple functions within a service or combined with other AWS services.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Topics

- [Scenarios](#)

Scenarios

Building an Amazon Lex chatbot

The following code example shows how to create a chatbot to engage your website visitors.

SDK for JavaScript (v3)

Shows how to use the Amazon Lex API to create a Chatbot within a web application to engage your web site visitors.

For complete source code and instructions on how to set up and run, see the full example [Building an Amazon Lex chatbot](#) in the AWS SDK for JavaScript developer guide.

Services used in this example

- Amazon Comprehend
- Amazon Lex
- Amazon Translate

Amazon MSK examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with Amazon MSK.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Topics

- [Serverless examples](#)

Serverless examples

Invoke a Lambda function from an Amazon MSK trigger

The following code example shows how to implement a Lambda function that receives an event triggered by receiving records from an Amazon MSK cluster. The function retrieves the MSK payload and logs the record contents.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [Serverless examples](#) repository.

Consuming an Amazon MSK event with Lambda using JavaScript.

```
exports.handler = async (event) => {
  // Iterate through keys
  for (let key in event.records) {
    console.log('Key: ', key)
    // Iterate through records
    event.records[key].map((record) => {
      console.log('Record: ', record)
      // Decode base64
      const msg = Buffer.from(record.value, 'base64').toString()
      console.log('Message:', msg)
    })
  }
}
```

Amazon Personalize examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with Amazon Personalize.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Topics

- [Actions](#)

Actions

CreateBatchInferenceJob

The following code example shows how to use CreateBatchInferenceJob.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// Get service clients module and commands using ES6 syntax.
import { CreateBatchInferenceJobCommand } from
  "@aws-sdk/client-personalize";
import { personalizeClient } from "../libs/personalizeClients.js";

// Or, create the client here.
// const personalizeClient = new PersonalizeClient({ region: "REGION"});

// Set the batch inference job's parameters.

export const createBatchInferenceJobParam = {
  jobName: 'JOB_NAME',
  jobInput: { /* required */
    s3DataSource: { /* required */
      path: 'INPUT_PATH', /* required */
      // kmsKeyArn: 'INPUT_KMS_KEY_ARN' /* optional */
    }
  },
  jobOutput: { /* required */
    s3DataDestination: { /* required */
      path: 'OUTPUT_PATH', /* required */
      // kmsKeyArn: 'OUTPUT_KMS_KEY_ARN' /* optional */
    }
  },
  roleArn: 'ROLE_ARN', /* required */
  solutionVersionArn: 'SOLUTION_VERSION_ARN', /* required */
  numResults: 20 /* optional integer*/
};

export const run = async () => {
  try {
    const response = await personalizeClient.send(new
      CreateBatchInferenceJobCommand(createBatchInferenceJobParam));
    console.log("Success", response);
  }
}
```

```
    return response; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

- For API details, see [CreateBatchInferenceJob](#) in *AWS SDK for JavaScript API Reference*.

CreateBatchSegmentJob

The following code example shows how to use `CreateBatchSegmentJob`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// Get service clients module and commands using ES6 syntax.
import { CreateBatchSegmentJobCommand } from
  "@aws-sdk/client-personalize";
import { personalizeClient } from "../libs/personalizeClients.js";

// Or, create the client here.
// const personalizeClient = new PersonalizeClient({ region: "REGION"});

// Set the batch segment job's parameters.

export const createBatchSegmentJobParam = {
  jobName: 'NAME',
  jobInput: { /* required */
    s3DataSource: { /* required */
      path: 'INPUT_PATH', /* required */
      // kmsKeyArn: 'INPUT_KMS_KEY_ARN' /* optional */
    }
  },
},
```

```

jobOutput: {          /* required */
  s3DataDestination: { /* required */
    path: 'OUTPUT_PATH', /* required */
    // kmsKeyArn: 'OUTPUT_KMS_KEY_ARN' /* optional */
  }
},
roleArn: 'ROLE_ARN', /* required */
solutionVersionArn: 'SOLUTION_VERSION_ARN', /* required */
numResults: 20 /* optional */
};

export const run = async () => {
  try {
    const response = await personalizeClient.send(new
CreateBatchSegmentJobCommand(createBatchSegmentJobParam));
    console.log("Success", response);
    return response; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
run();

```

- For API details, see [CreateBatchSegmentJob](#) in *AWS SDK for JavaScript API Reference*.

CreateCampaign

The following code example shows how to use CreateCampaign.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```

// Get service clients module and commands using ES6 syntax.

import { CreateCampaignCommand } from
"@aws-sdk/client-personalize";

```

```
import { personalizeClient } from "../libs/personalizeClients.js";

// Or, create the client here.
// const personalizeClient = new PersonalizeClient({ region: "REGION"});

// Set the campaign's parameters.
export const createCampaignParam = {
  solutionVersionArn: 'SOLUTION_VERSION_ARN', /* required */
  name: 'NAME', /* required */
  minProvisionedTPS: 1 /* optional integer */
}

export const run = async () => {
  try {
    const response = await personalizeClient.send(new
    CreateCampaignCommand(createCampaignParam));
    console.log("Success", response);
    return response; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

- For API details, see [CreateCampaign](#) in *AWS SDK for JavaScript API Reference*.

CreateDataset

The following code example shows how to use CreateDataset.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// Get service clients module and commands using ES6 syntax.
import { CreateDatasetCommand } from
"@aws-sdk/client-personalize";
```

```
import { personalizeClient } from "../libs/personalizeClients.js";

// Or, create the client here.
// const personalizeClient = new PersonalizeClient({ region: "REGION"});

// Set the dataset's parameters.
export const createDatasetParam = {
  datasetGroupArn: 'DATASET_GROUP_ARN', /* required */
  datasetType: 'DATASET_TYPE', /* required */
  name: 'NAME', /* required */
  schemaArn: 'SCHEMA_ARN' /* required */
}

export const run = async () => {
  try {
    const response = await personalizeClient.send(new
    CreateDatasetCommand(createDatasetParam));
    console.log("Success", response);
    return response; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

- For API details, see [CreateDataset](#) in *AWS SDK for JavaScript API Reference*.

CreateDatasetExportJob

The following code example shows how to use `CreateDatasetExportJob`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// Get service clients module and commands using ES6 syntax.
```

```
import { CreateDatasetExportJobCommand } from
  "@aws-sdk/client-personalize";
import { personalizeClient } from "../libs/personalizeClients.js";

// Or, create the client here.
// const personalizeClient = new PersonalizeClient({ region: "REGION"});

// Set the export job parameters.
export const datasetExportJobParam = {
  datasetArn: 'DATASET_ARN', /* required */
  jobOutput: {
    s3DataDestination: {
      path: 'S3_DESTINATION_PATH' /* required */
      //kmsKeyArn: 'ARN' /* include if your bucket uses AWS KMS for encryption
    }
  },
  jobName: 'NAME', /* required */
  roleArn: 'ROLE_ARN' /* required */
}

export const run = async () => {
  try {
    const response = await personalizeClient.send(new
    CreateDatasetExportJobCommand(datasetExportJobParam));
    console.log("Success", response);
    return response; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

- For API details, see [CreateDatasetExportJob](#) in *AWS SDK for JavaScript API Reference*.

CreateDatasetGroup

The following code example shows how to use CreateDatasetGroup.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// Get service clients module and commands using ES6 syntax.

import { CreateDatasetGroupCommand } from "@aws-sdk/client-personalize";
import { personalizeClient } from "../libs/personalizeClients.js";

// Or, create the client here.
// const personalizeClient = new PersonalizeClient({ region: "REGION"});

// Set the dataset group parameters.
export const createDatasetGroupParam = {
  name: "NAME" /* required */,
};

export const run = async (createDatasetGroupParam) => {
  try {
    const response = await personalizeClient.send(
      new CreateDatasetGroupCommand(createDatasetGroupParam),
    );
    console.log("Success", response);
    return "Run successfully"; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
run(createDatasetGroupParam);
```

Create a domain dataset group.

```
// Get service clients module and commands using ES6 syntax.
import { CreateDatasetGroupCommand } from
  "@aws-sdk/client-personalize";
import { personalizeClient } from "../libs/personalizeClients.js";
```



```
// Or, create the client here.
// const personalizeClient = new PersonalizeClient({ region: "REGION"});

// Set the domain dataset group parameters.
export const domainDatasetGroupParams = {
  name: 'NAME', /* required */
  domain: 'DOMAIN' /* required for a domain dsG, specify ECOMMERCE or
  VIDEO_ON_DEMAND */
}

export const run = async () => {
  try {
    const response = await personalizeClient.send(new
    CreateDatasetGroupCommand(domainDatasetGroupParams));
    console.log("Success", response);
    return response; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

- For API details, see [CreateDatasetGroup](#) in *AWS SDK for JavaScript API Reference*.

CreateDatasetImportJob

The following code example shows how to use `CreateDatasetImportJob`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// Get service clients module and commands using ES6 syntax.
import {CreateDatasetImportJobCommand } from
  "@aws-sdk/client-personalize";
import { personalizeClient } from "../libs/personalizeClients.js";
```

```
// Or, create the client here.
// const personalizeClient = new PersonalizeClient({ region: "REGION"});

// Set the dataset import job parameters.
export const datasetImportJobParam = {
  datasetArn: 'DATASET_ARN', /* required */
  dataSource: { /* required */
    dataLocation: 'S3_PATH'
  },
  jobName: 'NAME', /* required */
  roleArn: 'ROLE_ARN' /* required */
}

export const run = async () => {
  try {
    const response = await personalizeClient.send(new
    CreateDatasetImportJobCommand(datasetImportJobParam));
    console.log("Success", response);
    return response; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

- For API details, see [CreateDatasetImportJob](#) in *AWS SDK for JavaScript API Reference*.

CreateEventTracker

The following code example shows how to use CreateEventTracker.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// Get service clients module and commands using ES6 syntax.
import { CreateEventTrackerCommand } from
```

```
"@aws-sdk/client-personalize";
import { personalizeClient } from "../libs/personalizeClients.js";

// Or, create the client here.
// const personalizeClient = new PersonalizeClient({ region: "REGION"});

// Set the event tracker's parameters.
export const createEventTrackerParam = {
  datasetGroupArn: 'DATASET_GROUP_ARN', /* required */
  name: 'NAME', /* required */
}

export const run = async () => {
  try {
    const response = await personalizeClient.send(new
    CreateEventTrackerCommand(createEventTrackerParam));
    console.log("Success", response);
    return response; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

- For API details, see [CreateEventTracker](#) in *AWS SDK for JavaScript API Reference*.

CreateFilter

The following code example shows how to use `CreateFilter`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// Get service clients module and commands using ES6 syntax.
import { CreateFilterCommand } from
"@aws-sdk/client-personalize";
```

```
import { personalizeClient } from "../libs/personalizeClients.js";
// Or, create the client here.
// const personalizeClient = new PersonalizeClient({ region: "REGION"});

// Set the filter's parameters.
export const createFilterParam = {
  datasetGroupArn: 'DATASET_GROUP_ARN', /* required */
  name: 'NAME', /* required */
  filterExpression: 'FILTER_EXPRESSION' /*required */
}

export const run = async () => {
  try {
    const response = await personalizeClient.send(new
CreateFilterCommand(createFilterParam));
    console.log("Success", response);
    return response; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

- For API details, see [CreateFilter](#) in *AWS SDK for JavaScript API Reference*.

CreateRecommender

The following code example shows how to use CreateRecommender.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// Get service clients module and commands using ES6 syntax.
import { CreateRecommenderCommand } from
"@aws-sdk/client-personalize";
import { personalizeClient } from "../libs/personalizeClients.js";
```

```
// Or, create the client here.
// const personalizeClient = new PersonalizeClient({ region: "REGION"});

// Set the recommender's parameters.
export const createRecommenderParam = {
  name: 'NAME', /* required */
  recipeArn: 'RECIPE_ARN', /* required */
  datasetGroupArn: 'DATASET_GROUP_ARN' /* required */
}

export const run = async () => {
  try {
    const response = await personalizeClient.send(new
    CreateRecommenderCommand(createRecommenderParam));
    console.log("Success", response);
    return response; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

- For API details, see [CreateRecommender](#) in *AWS SDK for JavaScript API Reference*.

CreateSchema

The following code example shows how to use CreateSchema.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// Get service clients module and commands using ES6 syntax.
import { CreateSchemaCommand } from
  "@aws-sdk/client-personalize";
import { personalizeClient } from "../libs/personalizeClients.js";
```

```
// Or, create the client here.
// const personalizeClient = new PersonalizeClient({ region: "REGION"});

import fs from 'fs';

let schemaFilePath = "SCHEMA_PATH";
let mySchema = "";

try {
  mySchema = fs.readFileSync(schemaFilePath).toString();
} catch (err) {
  mySchema = 'TEST' // For unit tests.
}

// Set the schema parameters.
export const createSchemaParam = {
  name: 'NAME', /* required */
  schema: mySchema /* required */
};

export const run = async () => {
  try {
    const response = await personalizeClient.send(new
    CreateSchemaCommand(createSchemaParam));
    console.log("Success", response);
    return response; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

Create a schema with a domain.

```
// Get service clients module and commands using ES6 syntax.
import { CreateSchemaCommand } from
  "@aws-sdk/client-personalize";
import { personalizeClient } from "../libs/personalizeClients.js";

// Or, create the client here.
// const personalizeClient = new PersonalizeClient({ region: "REGION"});
```

```
import fs from 'fs';

let schemaFilePath = "SCHEMA_PATH";
let mySchema = "";

try {
  mySchema = fs.readFileSync(schemaFilePath).toString();
} catch (err) {
  mySchema = 'TEST' // for unit tests.
}

// Set the domain schema parameters.
export const createDomainSchemaParam = {
  name: 'NAME', /* required */
  schema: mySchema, /* required */
  domain: 'DOMAIN' /* required for a domain dataset group, specify ECOMMERCE or
  VIDEO_ON_DEMAND */
};

export const run = async () => {
  try {
    const response = await personalizeClient.send(new
    CreateSchemaCommand(createDomainSchemaParam));
    console.log("Success", response);
    return response; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

- For API details, see [CreateSchema](#) in *AWS SDK for JavaScript API Reference*.

CreateSolution

The following code example shows how to use CreateSolution.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// Get service clients module and commands using ES6 syntax.
import { CreateSolutionCommand } from
  "@aws-sdk/client-personalize";
import { personalizeClient } from "../libs/personalizeClients.js";
// Or, create the client here.
// const personalizeClient = new PersonalizeClient({ region: "REGION"});

// Set the solution parameters.
export const createSolutionParam = {
  datasetGroupArn: 'DATASET_GROUP_ARN', /* required */
  recipeArn: 'RECIPE_ARN', /* required */
  name: 'NAME' /* required */
}

export const run = async () => {
  try {
    const response = await personalizeClient.send(new
    CreateSolutionCommand(createSolutionParam));
    console.log("Success", response);
    return response; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

- For API details, see [CreateSolution](#) in *AWS SDK for JavaScript API Reference*.

CreateSolutionVersion

The following code example shows how to use `CreateSolutionVersion`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// Get service clients module and commands using ES6 syntax.
import { CreateSolutionVersionCommand } from
  "@aws-sdk/client-personalize";
import { personalizeClient } from "../libs/personalizeClients.js";
// Or, create the client here.
// const personalizeClient = new PersonalizeClient({ region: "REGION"});

// Set the solution version parameters.
export const solutionVersionParam = {
  solutionArn: 'SOLUTION_ARN' /* required */
}

export const run = async () => {
  try {
    const response = await personalizeClient.send(new
    CreateSolutionVersionCommand(solutionVersionParam));
    console.log("Success", response);
    return response; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

- For API details, see [CreateSolutionVersion](#) in *AWS SDK for JavaScript API Reference*.

Amazon Personalize Events examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with Amazon Personalize Events.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Topics

- [Actions](#)

Actions

PutEvents

The following code example shows how to use PutEvents.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// Get service clients module and commands using ES6 syntax.
import { PutEventsCommand } from "@aws-sdk/client-personalize-events";
import { personalizeEventsClient } from "../libs/personalizeClients.js";
// Or, create the client here.
// const personalizeEventsClient = new PersonalizeEventsClient({ region: "REGION"});

// Convert your UNIX timestamp to a Date.
const sentAtDate = new Date(1613443801 * 1000); // 1613443801 is a testing value.
Replace it with your sentAt timestamp in UNIX format.

// Set put events parameters.
var putEventsParam = {
  eventList: [
    /* required */
    {
      eventType: "EVENT_TYPE" /* required */,
      sentAt: sentAtDate /* required, must be a Date with js */,
    }
  ]
}
```

```
    eventId: "EVENT_ID" /* optional */,
    itemId: "ITEM_ID" /* optional */,
  },
],
sessionId: "SESSION_ID" /* required */,
trackingId: "TRACKING_ID" /* required */,
userId: "USER_ID" /* required */,
};
export const run = async () => {
  try {
    const response = await personalizeEventsClient.send(
      new PutEventsCommand(putEventsParam),
    );
    console.log("Success!", response);
    return response; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

- For API details, see [PutEvents](#) in *AWS SDK for JavaScript API Reference*.

PutItems

The following code example shows how to use PutItems.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// Get service clients module and commands using ES6 syntax.
import { PutItemsCommand } from "@aws-sdk/client-personalize-events";
import { personalizeEventsClient } from "../libs/personalizeClients.js";
// Or, create the client here.
// const personalizeEventsClient = new PersonalizeEventsClient({ region: "REGION"});
```

```
// Set the put items parameters. For string properties and values, use the \
character to escape quotes.
var putItemsParam = {
  datasetArn: "DATASET_ARN" /* required */,
  items: [
    /* required */
    {
      itemId: "ITEM_ID" /* required */,
      properties:
        '{"PROPERTY1_NAME": "PROPERTY1_VALUE", "PROPERTY2_NAME": "PROPERTY2_VALUE",
"PROPERTY3_NAME": "PROPERTY3_VALUE"}' /* optional */,
    },
  ],
};
export const run = async () => {
  try {
    const response = await personalizeEventsClient.send(
      new PutItemsCommand(putItemsParam),
    );
    console.log("Success!", response);
    return response; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

- For API details, see [PutItems](#) in *AWS SDK for JavaScript API Reference*.

PutUsers

The following code example shows how to use PutUsers.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// Get service clients module and commands using ES6 syntax.
import { PutUsersCommand } from "@aws-sdk/client-personalize-events";
import { personalizeEventsClient } from "../libs/personalizeClients.js";
// Or, create the client here.
// const personalizeEventsClient = new PersonalizeEventsClient({ region: "REGION"});

// Set the put users parameters. For string properties and values, use the \
character to escape quotes.
var putUsersParam = {
  datasetArn: "DATASET_ARN",
  users: [
    {
      userId: "USER_ID",
      properties: '{"PROPERTY1_NAME": "PROPERTY1_VALUE"}',
    },
  ],
};
export const run = async () => {
  try {
    const response = await personalizeEventsClient.send(
      new PutUsersCommand(putUsersParam),
    );
    console.log("Success!", response);
    return response; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

- For API details, see [PutUsers](#) in *AWS SDK for JavaScript API Reference*.

Amazon Personalize Runtime examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with Amazon Personalize Runtime.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Topics

- [Actions](#)

Actions

GetPersonalizedRanking

The following code example shows how to use `GetPersonalizedRanking`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// Get service clients module and commands using ES6 syntax.
import { GetPersonalizedRankingCommand } from
  "@aws-sdk/client-personalize-runtime";
import { personalizeRuntimeClient } from "../libs/personalizeClients.js";
// Or, create the client here.
// const personalizeRuntimeClient = new PersonalizeRuntimeClient({ region:
  "REGION"});

// Set the ranking request parameters.
export const getPersonalizedRankingParam = {
  campaignArn: "CAMPAIGN_ARN", /* required */
  userId: 'USER_ID', /* required */
  inputList: ["ITEM_ID_1", "ITEM_ID_2", "ITEM_ID_3", "ITEM_ID_4"]
}

export const run = async () => {
  try {
    const response = await personalizeRuntimeClient.send(new
      GetPersonalizedRankingCommand(getPersonalizedRankingParam));
    console.log("Success!", response);
  }
}
```

```
    return response; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

- For API details, see [GetPersonalizedRanking](#) in *AWS SDK for JavaScript API Reference*.

GetRecommendations

The following code example shows how to use `GetRecommendations`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// Get service clients module and commands using ES6 syntax.
import { GetRecommendationsCommand } from
  "@aws-sdk/client-personalize-runtime";

import { personalizeRuntimeClient } from "../libs/personalizeClients.js";
// Or, create the client here.
// const personalizeRuntimeClient = new PersonalizeRuntimeClient({ region:
  "REGION"});

// Set the recommendation request parameters.
export const getRecommendationsParam = {
  campaignArn: 'CAMPAIGN_ARN', /* required */
  userId: 'USER_ID',          /* required */
  numResults: 15             /* optional */
}

export const run = async () => {
  try {
```

```
    const response = await personalizeRuntimeClient.send(new
  GetRecommendationsCommand(getRecommendationsParam));
    console.log("Success!", response);
    return response; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

Get recommendation with a filter (custom dataset group).

```
// Get service clients module and commands using ES6 syntax.
import { GetRecommendationsCommand } from
  "@aws-sdk/client-personalize-runtime";
import { personalizeRuntimeClient } from "./libs/personalizeClients.js";
// Or, create the client here.
// const personalizeRuntimeClient = new PersonalizeRuntimeClient({ region:
  "REGION"});

// Set the recommendation request parameters.
export const getRecommendationsParam = {
  recommenderArn: 'RECOMMENDER_ARN', /* required */
  userId: 'USER_ID', /* required */
  numResults: 15 /* optional */
}

export const run = async () => {
  try {
    const response = await personalizeRuntimeClient.send(new
  GetRecommendationsCommand(getRecommendationsParam));
    console.log("Success!", response);
    return response; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

Get filtered recommendations from a recommender created in a domain dataset group.


```
// Get service clients module and commands using ES6 syntax.
import { GetRecommendationsCommand } from
  "@aws-sdk/client-personalize-runtime";
import { personalizeRuntimeClient } from "./libs/personalizeClients.js";
// Or, create the client here:
// const personalizeRuntimeClient = new PersonalizeRuntimeClient({ region:
  "REGION"});

// Set recommendation request parameters.
export const getRecommendationsParam = {
  campaignArn: 'CAMPAIGN_ARN', /* required */
  userId: 'USER_ID',          /* required */
  numResults: 15,             /* optional */
  filterArn: 'FILTER_ARN',    /* required to filter recommendations */
  filterValues: {
    "PROPERTY": "\"VALUE\"" /* Only required if your filter has a placeholder
parameter */
  }
}

export const run = async () => {
  try {
    const response = await personalizeRuntimeClient.send(new
  GetRecommendationsCommand(getRecommendationsParam));
    console.log("Success!", response);
    return response; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

- For API details, see [GetRecommendations](#) in *AWS SDK for JavaScript API Reference*.

Amazon Pinpoint examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with Amazon Pinpoint.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Topics

- [Actions](#)

Actions

SendMessages

The following code example shows how to use SendMessages.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create the client in a separate module and export it.

```
import { PinpointClient } from "@aws-sdk/client-pinpoint";
// Set the AWS Region.
const REGION = "us-east-1";
export const pinClient = new PinpointClient({ region: REGION });
```

Send an email message.

```
// Import required AWS SDK clients and commands for Node.js
import { SendMessagesCommand } from "@aws-sdk/client-pinpoint";
import { pinClient } from "../libs/pinClient.js";

// The FromAddress must be verified in SES.
const fromAddress = "FROM_ADDRESS";
const toAddress = "TO_ADDRESS";
const projectId = "PINPOINT_PROJECT_ID";

// The subject line of the email.
var subject = "Amazon Pinpoint Test (AWS SDK for JavaScript in Node.js)";
```

```
// The email body for recipients with non-HTML email clients.
var body_text = `Amazon Pinpoint Test (SDK for JavaScript in Node.js)
-----
This email was sent with Amazon Pinpoint using the AWS SDK for JavaScript in
Node.js.
For more information, see https://aws.amazon.com/sdk-for-node-js/`;

// The body of the email for recipients whose email clients support HTML content.
var body_html = `
<head></head>
<body>
  <h1>Amazon Pinpoint Test (SDK for JavaScript in Node.js)</h1>
  <p>This email was sent with
    <a href='https://aws.amazon.com/pinpoint/'>the Amazon Pinpoint Email API</a>
using the
    <a href='https://aws.amazon.com/sdk-for-node-js/'>
      AWS SDK for JavaScript in Node.js</a>.</p>
</body>
</html>`;

// The character encoding for the subject line and message body of the email.
var charset = "UTF-8";

const params = {
  ApplicationId: projectId,
  MessageRequest: {
    Addresses: {
      [toAddress]: {
        ChannelType: "EMAIL",
      },
    },
    MessageConfiguration: {
      EmailMessage: {
        FromAddress: fromAddress,
        SimpleEmail: {
          Subject: {
            Charset: charset,
            Data: subject,
          },
          HtmlPart: {
            Charset: charset,
            Data: body_html,
          },
        },
      },
    },
  },
}
```

```
        TextPart: {
          Charset: charset,
          Data: body_text,
        },
      },
    },
  },
};

const run = async () => {
  try {
    const { MessageResponse } = await pinClient.send(
      new SendMessagesCommand(params),
    );

    if (!MessageResponse) {
      throw new Error("No message response.");
    }

    if (!MessageResponse.Result) {
      throw new Error("No message result.");
    }

    const recipientResult = MessageResponse.Result[toAddress];

    if (recipientResult.StatusCode !== 200) {
      throw new Error(recipientResult.StatusMessage);
    } else {
      console.log(recipientResult.MessageId);
    }
  } catch (err) {
    console.log(err.message);
  }
};

run();
```

Send an SMS message.

```
// Import required AWS SDK clients and commands for Node.js
```

```
import { SendMessagesCommand } from "@aws-sdk/client-pinpoint";
import { pinClient } from "../libs/pinClient.js";

/* The phone number or short code to send the message from. The phone number
   or short code that you specify has to be associated with your Amazon Pinpoint
   account. For best results, specify long codes in E.164 format. */
const originationNumber = "SENDER_NUMBER"; //e.g., +1XXXXXXXXXX

// The recipient's phone number. For best results, you should specify the phone
   number in E.164 format.
const destinationNumber = "RECEIVER_NUMBER"; //e.g., +1XXXXXXXXXX

// The content of the SMS message.
const message =
  "This message was sent through Amazon Pinpoint " +
  "using the AWS SDK for JavaScript in Node.js. Reply STOP to " +
  "opt out.";

/*The Amazon Pinpoint project/application ID to use when you send this message.
   Make sure that the SMS channel is enabled for the project or application
   that you choose.*/
const projectId = "PINPOINT_PROJECT_ID"; //e.g., XXXXXXXX66e4e9986478cXXXXXXXXXX

/* The type of SMS message that you want to send. If you plan to send
   time-sensitive content, specify TRANSACTIONAL. If you plan to send
   marketing-related content, specify PROMOTIONAL.*/
var messageType = "TRANSACTIONAL";

// The registered keyword associated with the originating short code.
var registeredKeyword = "myKeyword";

/* The sender ID to use when sending the message. Support for sender ID
   // varies by country or region. For more information, see
   https://docs.aws.amazon.com/pinpoint/latest/userguide/channels-sms-countries.html.*/

var senderId = "MySenderID";

// Specify the parameters to pass to the API.
var params = {
  ApplicationId: projectId,
  MessageRequest: {
    Addresses: {
      [destinationNumber]: {
        ChannelType: "SMS",
```

```
    },
  },
  MessageConfiguration: {
    SMSMessage: {
      Body: message,
      Keyword: registeredKeyword,
      MessageType: messageType,
      OriginationNumber: originationNumber,
      SenderId: senderId,
    },
  },
},
};

const run = async () => {
  try {
    const data = await pinClient.send(new SendMessagesCommand(params));
    console.log(
      "Message sent! " +
      data["MessageResponse"]["Result"][destinationNumber]["StatusMessage"],
    );
  } catch (err) {
    console.log(err);
  }
};

run();
```

- For API details, see [SendMessages](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Send an email message.

```
"use strict";
```

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

// The AWS Region that you want to use to send the email. For a list of
// AWS Regions where the Amazon Pinpoint API is available, see
// https://docs.aws.amazon.com/pinpoint/latest/apireference/
const aws_region = "us-west-2";

// The "From" address. This address has to be verified in Amazon Pinpoint
// in the region that you use to send email.
const senderAddress = "sender@example.com";

// The address on the "To" line. If your Amazon Pinpoint account is in
// the sandbox, this address also has to be verified.
var toAddress = "recipient@example.com";

// The Amazon Pinpoint project/application ID to use when you send this message.
// Make sure that the SMS channel is enabled for the project or application
// that you choose.
const appId = "ce796be37f32f178af652b26eexample";

// The subject line of the email.
var subject = "Amazon Pinpoint (AWS SDK for JavaScript in Node.js)";

// The email body for recipients with non-HTML email clients.
var body_text = `Amazon Pinpoint Test (SDK for JavaScript in Node.js)
-----
This email was sent with Amazon Pinpoint using the AWS SDK for JavaScript in
Node.js.
For more information, see https://aws.amazon.com/sdk-for-node-js/`;

// The body of the email for recipients whose email clients support HTML content.
var body_html = `
<head></head>
<body>
  <h1>Amazon Pinpoint Test (SDK for JavaScript in Node.js)</h1>
  <p>This email was sent with
    <a href='https://aws.amazon.com/pinpoint/'>the Amazon Pinpoint API</a> using the
    <a href='https://aws.amazon.com/sdk-for-node-js/'>
      AWS SDK for JavaScript in Node.js</a>.</p>
</body>
</html>`;

// The character encoding the you want to use for the subject line and
// message body of the email.
```

```
var charset = "UTF-8";

// Specify that you're using a shared credentials file.
var credentials = new AWS.SharedIniFileCredentials({ profile: "default" });
AWS.config.credentials = credentials;

// Specify the region.
AWS.config.update({ region: aws_region });

//Create a new Pinpoint object.
var pinpoint = new AWS.Pinpoint();

// Specify the parameters to pass to the API.
var params = {
  ApplicationId: appId,
  MessageRequest: {
    Addresses: {
      [toAddress]: {
        ChannelType: "EMAIL",
      },
    },
    MessageConfiguration: {
      EmailMessage: {
        FromAddress: senderAddress,
        SimpleEmail: {
          Subject: {
            Charset: charset,
            Data: subject,
          },
          HtmlPart: {
            Charset: charset,
            Data: body_html,
          },
          TextPart: {
            Charset: charset,
            Data: body_text,
          },
        },
      },
    },
  },
};

//Try to send the email.
```



```
pinpoint.sendMessage(params, function (err, data) {
  // If something goes wrong, print an error message.
  if (err) {
    console.log(err.message);
  } else {
    console.log(
      "Email sent! Message ID: ",
      data["MessageResponse"]["Result"]["toAddress"]["MessageId"]
    );
  }
});
```

Send an SMS message.

```
"use strict";

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

// The AWS Region that you want to use to send the message. For a list of
// AWS Regions where the Amazon Pinpoint API is available, see
// https://docs.aws.amazon.com/pinpoint/latest/apireference/.
var aws_region = "us-east-1";

// The phone number or short code to send the message from. The phone number
// or short code that you specify has to be associated with your Amazon Pinpoint
// account. For best results, specify long codes in E.164 format.
var originationNumber = "+12065550199";

// The recipient's phone number. For best results, you should specify the
// phone number in E.164 format.
var destinationNumber = "+14255550142";

// The content of the SMS message.
var message =
  "This message was sent through Amazon Pinpoint " +
  "using the AWS SDK for JavaScript in Node.js. Reply STOP to " +
  "opt out.";

// The Amazon Pinpoint project/application ID to use when you send this message.
// Make sure that the SMS channel is enabled for the project or application
```

```
// that you choose.
var applicationId = "ce796be37f32f178af652b26eexample";

// The type of SMS message that you want to send. If you plan to send
// time-sensitive content, specify TRANSACTIONAL. If you plan to send
// marketing-related content, specify PROMOTIONAL.
var messageType = "TRANSACTIONAL";

// The registered keyword associated with the originating short code.
var registeredKeyword = "myKeyword";

// The sender ID to use when sending the message. Support for sender ID
// varies by country or region. For more information, see
// https://docs.aws.amazon.com/pinpoint/latest/userguide/channels-sms-countries.html
var senderId = "MySenderId";

// Specify that you're using a shared credentials file, and optionally specify
// the profile that you want to use.
var credentials = new AWS.SharedIniFileCredentials({ profile: "default" });
AWS.config.credentials = credentials;

// Specify the region.
AWS.config.update({ region: aws_region });

//Create a new Pinpoint object.
var pinpoint = new AWS.Pinpoint();

// Specify the parameters to pass to the API.
var params = {
  ApplicationId: applicationId,
  MessageRequest: {
    Addresses: {
      [destinationNumber]: {
        ChannelType: "SMS",
      },
    },
    MessageConfiguration: {
      SMSMessage: {
        Body: message,
        Keyword: registeredKeyword,
        MessageType: messageType,
        OriginationNumber: originationNumber,
        SenderId: senderId,
      },
    },
  },
}
```

```
    },
  },
};

//Try to send the message.
pinpoint.sendMessage(params, function (err, data) {
  // If something goes wrong, print an error message.
  if (err) {
    console.log(err.message);
    // Otherwise, show the unique ID for the message.
  } else {
    console.log(
      "Message sent! " +
      data["MessageResponse"]["Result"][destinationNumber]["StatusMessage"]
    );
  }
});
```

- For API details, see [SendMessage](#) in *AWS SDK for JavaScript API Reference*.

Amazon Polly examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with Amazon Polly.

Scenarios are code examples that show you how to accomplish specific tasks by calling multiple functions within a service or combined with other AWS services.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Topics

- [Scenarios](#)

Scenarios

Create an application to analyze customer feedback

The following code example shows how to create an application that analyzes customer comment cards, translates them from their original language, determines their sentiment, and generates an audio file from the translated text.

SDK for JavaScript (v3)

This example application analyzes and stores customer feedback cards. Specifically, it fulfills the need of a fictitious hotel in New York City. The hotel receives feedback from guests in various languages in the form of physical comment cards. That feedback is uploaded into the app through a web client. After an image of a comment card is uploaded, the following steps occur:

- Text is extracted from the image using Amazon Textract.
- Amazon Comprehend determines the sentiment of the extracted text and its language.
- The extracted text is translated to English using Amazon Translate.
- Amazon Polly synthesizes an audio file from the extracted text.

The full app can be deployed with the AWS CDK. For source code and deployment instructions, see the project in [GitHub](#). The following excerpts show how the AWS SDK for JavaScript is used inside of Lambda functions.

```
import {
  ComprehendClient,
  DetectDominantLanguageCommand,
  DetectSentimentCommand,
} from "@aws-sdk/client-comprehend";

/**
 * Determine the language and sentiment of the extracted text.
 *
 * @param {{ source_text: string }} extractTextOutput
 */
export const handler = async (extractTextOutput) => {
  const comprehendClient = new ComprehendClient({});

  const detectDominantLanguageCommand = new DetectDominantLanguageCommand({
    Text: extractTextOutput.source_text,
  });
```

```
// The source language is required for sentiment analysis and
// translation in the next step.
const { Languages } = await comprehendClient.send(
  detectDominantLanguageCommand,
);

const languageCode = Languages[0].LanguageCode;

const detectSentimentCommand = new DetectSentimentCommand({
  Text: extractTextOutput.source_text,
  LanguageCode: languageCode,
});

const { Sentiment } = await comprehendClient.send(detectSentimentCommand);

return {
  sentiment: Sentiment,
  language_code: languageCode,
};
};
```

```
import {
  DetectDocumentTextCommand,
  TextractClient,
} from "@aws-sdk/client-textract";

/**
 * Fetch the S3 object from the event and analyze it using Amazon Textract.
 *
 * @param {import("@types/aws-lambda").EventBridgeEvent<"Object Created">}
  eventBridgeS3Event
 */
export const handler = async (eventBridgeS3Event) => {
  const textractClient = new TextractClient();

  const detectDocumentTextCommand = new DetectDocumentTextCommand({
    Document: {
      S3Object: {
        Bucket: eventBridgeS3Event.bucket,
        Name: eventBridgeS3Event.object,
      },
    },
  },
};
```

```
});

// Textract returns a list of blocks. A block can be a line, a page, word, etc.
// Each block also contains geometry of the detected text.
// For more information on the Block type, see https://docs.aws.amazon.com/textract/latest/dg/API\_Block.html.
const { Blocks } = await textractClient.send(detectDocumentTextCommand);

// For the purpose of this example, we are only interested in words.
const extractedWords = Blocks.filter((b) => b.BlockType === "WORD").map(
  (b) => b.Text,
);

return extractedWords.join(" ");
};
```

```
import { PollyClient, SynthesizeSpeechCommand } from "@aws-sdk/client-polly";
import { S3Client } from "@aws-sdk/client-s3";
import { Upload } from "@aws-sdk/lib-storage";

/**
 * Synthesize an audio file from text.
 *
 * @param {{ bucket: string, translated_text: string, object: string}}
 * sourceDestinationConfig
 */
export const handler = async (sourceDestinationConfig) => {
  const pollyClient = new PollyClient({});

  const synthesizeSpeechCommand = new SynthesizeSpeechCommand({
    Engine: "neural",
    Text: sourceDestinationConfig.translated_text,
    VoiceId: "Ruth",
    OutputFormat: "mp3",
  });

  const { AudioStream } = await pollyClient.send(synthesizeSpeechCommand);

  const audioKey = `${sourceDestinationConfig.object}.mp3`;

  // Store the audio file in S3.
  const s3Client = new S3Client();
  const upload = new Upload({
```

```
    client: s3Client,
    params: {
      Bucket: sourceDestinationConfig.bucket,
      Key: audioKey,
      Body: AudioStream,
      ContentType: "audio/mp3",
    },
  });

  await upload.done();
  return audioKey;
};
```

```
import {
  TranslateClient,
  TranslateTextCommand,
} from "@aws-sdk/client-translate";

/**
 * Translate the extracted text to English.
 *
 * @param {{ extracted_text: string, source_language_code: string }}
 * textAndSourceLanguage
 */
export const handler = async (textAndSourceLanguage) => {
  const translateClient = new TranslateClient({});

  const translateCommand = new TranslateTextCommand({
    SourceLanguageCode: textAndSourceLanguage.source_language_code,
    TargetLanguageCode: "en",
    Text: textAndSourceLanguage.extracted_text,
  });

  const { TranslatedText } = await translateClient.send(translateCommand);

  return { translated_text: TranslatedText };
};
```

Services used in this example

- Amazon Comprehend
- Lambda

- Amazon Polly
- Amazon Textract
- Amazon Translate

Amazon RDS examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with Amazon RDS.

Scenarios are code examples that show you how to accomplish specific tasks by calling multiple functions within a service or combined with other AWS services.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Topics

- [Scenarios](#)
- [Serverless examples](#)

Scenarios

Create an Aurora Serverless work item tracker

The following code example shows how to create a web application that tracks work items in an Amazon Aurora Serverless database and uses Amazon Simple Email Service (Amazon SES) to send reports.

SDK for JavaScript (v3)

Shows how to use the AWS SDK for JavaScript (v3) to create a web application that tracks work items in an Amazon Aurora database and emails reports by using Amazon Simple Email Service (Amazon SES). This example uses a front end built with React.js to interact with an Express Node.js backend.

- Integrate a React.js web application with AWS services.
- List, add, and update items in an Aurora table.
- Send an email report of filtered work items by using Amazon SES.
- Deploy and manage example resources with the included AWS CloudFormation script.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- Aurora
- Amazon RDS
- Amazon RDS Data Service
- Amazon SES

Serverless examples

Connecting to an Amazon RDS database in a Lambda function

The following code example shows how to implement a Lambda function that connects to an RDS database. The function makes a simple database request and returns the result.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [Serverless examples](#) repository.

Connecting to an Amazon RDS database in a Lambda function using JavaScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
/*
Node.js code here.
*/
// ES6+ example
import { Signer } from "@aws-sdk/rds-signer";
import mysql from 'mysql2/promise';

async function createAuthToken() {
  // Define connection authentication parameters
  const dbinfo = {

    hostname: process.env.ProxyHostName,
```

```
    port: process.env.Port,
    username: process.env.DBUserName,
    region: process.env.AWS_REGION,
  }

  // Create RDS Signer object
  const signer = new Signer(dbinfo);

  // Request authorization token from RDS, specifying the username
  const token = await signer.getAuthToken();
  return token;
}

async function dbOps() {

  // Obtain auth token
  const token = await createAuthToken();
  // Define connection configuration
  let connectionConfig = {
    host: process.env.ProxyHostName,
    user: process.env.DBUserName,
    password: token,
    database: process.env.DBName,
    ssl: 'Amazon RDS'
  }
  // Create the connection to the DB
  const conn = await mysql.createConnection(connectionConfig);
  // Obtain the result of the query
  const [res,] = await conn.execute('select ?+? as sum', [3, 2]);
  return res;
}

export const handler = async (event) => {
  // Execute database flow
  const result = await dbOps();
  // Return result
  return {
    statusCode: 200,
    body: JSON.stringify("The selected sum is: " + result[0].sum)
  }
};
```

Connecting to an Amazon RDS database in a Lambda function using TypeScript.

```
import { Signer } from "@aws-sdk/rds-signer";
import mysql from 'mysql2/promise';

// RDS settings
// Using '!' (non-null assertion operator) to tell the TypeScript compiler that the
// DB settings are not null or undefined,
const proxy_host_name = process.env.PROXY_HOST_NAME!
const port = parseInt(process.env.PORT!)
const db_name = process.env.DB_NAME!
const db_user_name = process.env.DB_USER_NAME!
const aws_region = process.env.AWS_REGION!

async function createAuthToken(): Promise<string> {

    // Create RDS Signer object
    const signer = new Signer({
        hostname: proxy_host_name,
        port: port,
        region: aws_region,
        username: db_user_name
    });

    // Request authorization token from RDS, specifying the username
    const token = await signer.getAuthToken();
    return token;
}

async function dbOps(): Promise<mysql.QueryResult | undefined> {
    try {
        // Obtain auth token
        const token = await createAuthToken();
        const conn = await mysql.createConnection({
            host: proxy_host_name,
            user: db_user_name,
            password: token,
            database: db_name,
            ssl: 'Amazon RDS' // Ensure you have the CA bundle for SSL connection
        });
    }
}
```

```
    });
    const [rows, fields] = await conn.execute('SELECT ? + ? AS sum', [3, 2]);
    console.log('result:', rows);
    return rows;
  }
  catch (err) {
    console.log(err);
  }
}

export const lambdaHandler = async (event: any): Promise<{ statusCode: number; body:
string }> => {
  // Execute database flow
  const result = await dbOps();

  // Return error if result is undefined
  if (result == undefined)
    return {
      statusCode: 500,
      body: JSON.stringify(`Error with connection to DB host`)
    }

  // Return result
  return {
    statusCode: 200,
    body: JSON.stringify(`The selected sum is: ${result[0].sum}`)
  };
};
```

Amazon RDS Data Service examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with Amazon RDS Data Service.

Scenarios are code examples that show you how to accomplish specific tasks by calling multiple functions within a service or combined with other AWS services.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Topics

- [Scenarios](#)

Scenarios

Create an Aurora Serverless work item tracker

The following code example shows how to create a web application that tracks work items in an Amazon Aurora Serverless database and uses Amazon Simple Email Service (Amazon SES) to send reports.

SDK for JavaScript (v3)

Shows how to use the AWS SDK for JavaScript (v3) to create a web application that tracks work items in an Amazon Aurora database and emails reports by using Amazon Simple Email Service (Amazon SES). This example uses a front end built with React.js to interact with an Express Node.js backend.

- Integrate a React.js web application with AWS services.
- List, add, and update items in an Aurora table.
- Send an email report of filtered work items by using Amazon SES.
- Deploy and manage example resources with the included AWS CloudFormation script.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- Aurora
- Amazon RDS
- Amazon RDS Data Service
- Amazon SES

Amazon Redshift examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with Amazon Redshift.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Topics

- [Actions](#)

Actions

CreateCluster

The following code example shows how to use CreateCluster.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create the client.

```
import { RedshiftClient } from "@aws-sdk/client-redshift";
// Set the AWS Region.
const REGION = "REGION";
//Set the Redshift Service Object
const redshiftClient = new RedshiftClient({ region: REGION });
export { redshiftClient };
```

Create the cluster.

```
// Import required AWS SDK clients and commands for Node.js
import { CreateClusterCommand } from "@aws-sdk/client-redshift";
import { redshiftClient } from "./libs/redshiftClient.js";

const params = {
```

```
ClusterIdentifier: "CLUSTER_NAME", // Required
NodeType: "NODE_TYPE", //Required
MasterUsername: "MASTER_USER_NAME", // Required - must be lowercase
MasterUserPassword: "MASTER_USER_PASSWORD", // Required - must contain at least
one uppercase letter, and one number
ClusterType: "CLUSTER_TYPE", // Required
IAMRoleARN: "IAM_ROLE_ARN", // Optional - the ARN of an IAM role with permissions
your cluster needs to access other AWS services on your behalf, such as Amazon S3.
ClusterSubnetGroupName: "CLUSTER_SUBNET_GROUPNAME", //Optional - the name of a
cluster subnet group to be associated with this cluster. Defaults to 'default' if
not specified.
DBName: "DATABASE_NAME", // Optional - defaults to 'dev' if not specified
Port: "PORT_NUMBER", // Optional - defaults to '5439' if not specified
};

const run = async () => {
  try {
    const data = await redshiftClient.send(new CreateClusterCommand(params));
    console.log(
      "Cluster " + data.Cluster.ClusterIdentifier + " successfully created",
    );
    return data; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

- For API details, see [CreateCluster](#) in *AWS SDK for JavaScript API Reference*.

DeleteCluster

The following code example shows how to use DeleteCluster.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create the client.

```
import { RedshiftClient } from "@aws-sdk/client-redshift";
// Set the AWS Region.
const REGION = "REGION";
//Set the Redshift Service Object
const redshiftClient = new RedshiftClient({ region: REGION });
export { redshiftClient };
```

Create the cluster.

```
// Import required AWS SDK clients and commands for Node.js
import { DeleteClusterCommand } from "@aws-sdk/client-redshift";
import { redshiftClient } from "../libs/redshiftClient.js";

const params = {
  ClusterIdentifier: "CLUSTER_NAME",
  SkipFinalClusterSnapshot: false,
  FinalClusterSnapshotIdentifier: "CLUSTER_SNAPSHOT_ID",
};

const run = async () => {
  try {
    const data = await redshiftClient.send(new DeleteClusterCommand(params));
    console.log("Success, cluster deleted. ", data);
    return data; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

- For API details, see [DeleteCluster](#) in *AWS SDK for JavaScript API Reference*.

DescribeClusters

The following code example shows how to use DescribeClusters.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create the client.

```
import { RedshiftClient } from "@aws-sdk/client-redshift";
// Set the AWS Region.
const REGION = "REGION";
//Set the Redshift Service Object
const redshiftClient = new RedshiftClient({ region: REGION });
export { redshiftClient };
```

Describe your clusters.

```
// Import required AWS SDK clients and commands for Node.js
import { DescribeClustersCommand } from "@aws-sdk/client-redshift";
import { redshiftClient } from "../libs/redshiftClient.js";

const params = {
  ClusterIdentifier: "CLUSTER_NAME",
};

const run = async () => {
  try {
    const data = await redshiftClient.send(new DescribeClustersCommand(params));
    console.log("Success", data);
    return data; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

- For API details, see [DescribeClusters](#) in *AWS SDK for JavaScript API Reference*.

ModifyCluster

The following code example shows how to use `ModifyCluster`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create the client.

```
import { RedshiftClient } from "@aws-sdk/client-redshift";
// Set the AWS Region.
const REGION = "REGION";
//Set the Redshift Service Object
const redshiftClient = new RedshiftClient({ region: REGION });
export { redshiftClient };
```

Modify a cluster.

```
// Import required AWS SDK clients and commands for Node.js
import { ModifyClusterCommand } from "@aws-sdk/client-redshift";
import { redshiftClient } from "../libs/redshiftClient.js";

// Set the parameters
const params = {
  ClusterIdentifier: "CLUSTER_NAME",
  MasterUserPassword: "NEW_MASTER_USER_PASSWORD",
};

const run = async () => {
  try {
    const data = await redshiftClient.send(new ModifyClusterCommand(params));
    console.log("Success was modified.", data);
    return data; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
}
```

```
};  
run();
```

- For API details, see [ModifyCluster](#) in *AWS SDK for JavaScript API Reference*.

Amazon Rekognition examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with Amazon Rekognition.

Scenarios are code examples that show you how to accomplish specific tasks by calling multiple functions within a service or combined with other AWS services.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Topics

- [Scenarios](#)

Scenarios

Create a serverless application to manage photos

The following code example shows how to create a serverless application that lets users manage photos using labels.

SDK for JavaScript (v3)

Shows how to develop a photo asset management application that detects labels in images using Amazon Rekognition and stores them for later retrieval.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

For a deep dive into the origin of this example see the post on [AWS Community](#).

Services used in this example

- API Gateway

- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Detect PPE in images

The following code example shows how to build an app that uses Amazon Rekognition to detect Personal Protective Equipment (PPE) in images.

SDK for JavaScript (v3)

Shows how to use Amazon Rekognition with the AWS SDK for JavaScript to create an application to detect personal protective equipment (PPE) in images located in an Amazon Simple Storage Service (Amazon S3) bucket. The app saves the results to an Amazon DynamoDB table, and sends the admin an email notification with the results using Amazon Simple Email Service (Amazon SES).

Learn how to:

- Create an unauthenticated user using Amazon Cognito.
- Analyze images for PPE using Amazon Rekognition.
- Verify an email address for Amazon SES.
- Update a DynamoDB table with results.
- Send an email notification using Amazon SES.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- DynamoDB
- Amazon Rekognition
- Amazon S3
- Amazon SES

Detect objects in images

The following code example shows how to build an app that uses Amazon Rekognition to detect objects by category in images.

SDK for JavaScript (v3)

Shows how to use Amazon Rekognition with the AWS SDK for JavaScript to create an app that uses Amazon Rekognition to identify objects by category in images located in an Amazon Simple Storage Service (Amazon S3) bucket. The app sends the admin an email notification with the results using Amazon Simple Email Service (Amazon SES).

Learn how to:

- Create an unauthenticated user using Amazon Cognito.
- Analyze images for objects using Amazon Rekognition.
- Verify an email address for Amazon SES.
- Send an email notification using Amazon SES.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- Amazon Rekognition
- Amazon S3
- Amazon SES

Detect people and objects in a video

The following code example shows how to detect people and objects in a video with Amazon Rekognition.

SDK for JavaScript (v3)

Shows how to use Amazon Rekognition with the AWS SDK for JavaScript to create an app to detect faces and objects in videos located in an Amazon Simple Storage Service (Amazon S3) bucket. The app sends the admin an email notification with the results using Amazon Simple Email Service (Amazon SES).

Learn how to:

- Create an unauthenticated user using Amazon Cognito.
- Analyze images for PPE using Amazon Rekognition.
- Verify an email address for Amazon SES.
- Send an email notification using Amazon SES.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- Amazon Rekognition
- Amazon S3
- Amazon SES

Amazon S3 examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with Amazon S3.

Basics are code examples that show you how to perform the essential operations within a service.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Scenarios are code examples that show you how to accomplish specific tasks by calling multiple functions within a service or combined with other AWS services.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Get started

Hello Amazon S3

The following code examples show how to get started using Amazon S3.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { ListBucketsCommand, S3Client } from "@aws-sdk/client-s3";

// When no region or credentials are provided, the SDK will use the
// region and credentials from the local AWS config.
const client = new S3Client({});

export const helloS3 = async () => {
  const command = new ListBucketsCommand({});

  const { Buckets } = await client.send(command);
  console.log("Buckets: ");
  console.log(Buckets.map((bucket) => bucket.Name).join("\n"));
  return Buckets;
};
```

- For API details, see [ListBuckets](#) in *AWS SDK for JavaScript API Reference*.

Topics

- [Basics](#)
- [Actions](#)
- [Scenarios](#)
- [Serverless examples](#)

Basics

Learn the basics

The following code example shows how to:

- Create a bucket and upload a file to it.

- Download an object from a bucket.
- Copy an object to a subfolder in a bucket.
- List the objects in a bucket.
- Delete the bucket objects and the bucket.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

First, import all the necessary modules.

```
// Used to check if currently running file is this file.
import { fileURLToPath } from "url";
import { readdirSync, readFileSync, writeFileSync } from "fs";

// Local helper utils.
import { dirnameFromMetaUrl } from "@aws-doc-sdk-examples/lib/utils/util-fs.js";
import { Prompter } from "@aws-doc-sdk-examples/lib/prompter.js";
import { wrapText } from "@aws-doc-sdk-examples/lib/utils/util-string.js";

import {
  S3Client,
  CreateBucketCommand,
  PutObjectCommand,
  ListObjectsCommand,
  CopyObjectCommand,
  GetObjectCommand,
  DeleteObjectsCommand,
  DeleteBucketCommand,
} from "@aws-sdk/client-s3";
```

The preceding imports reference some helper utilities. These utilities are local to the GitHub repository linked at the start of this section. For your reference, see the following implementations of those utilities.


```
export const dirnameFromMetaUrl = (metaUrl) =>
  fileURLToPath(new URL(".", metaUrl));

import { select, input, confirm, checkbox } from "@inquirer/prompts";

export class Prompter {
  /**
   * @param {{ message: string, choices: { name: string, value: string }[] }} options
   */
  select(options) {
    return select(options);
  }

  /**
   * @param {{ message: string }} options
   */
  input(options) {
    return input(options);
  }

  /**
   * @param {string} prompt
   */
  checkContinue = async (prompt = "") => {
    const prefix = prompt && prompt + " ";
    let ok = await this.confirm({
      message: `${prefix}Continue?`,
    });
    if (!ok) throw new Error("Exiting...");
  };

  /**
   * @param {{ message: string }} options
   */
  confirm(options) {
    return confirm(options);
  }

  /**
   * @param {{ message: string, choices: { name: string, value: string }[] }} options
   */
  checkbox(options) {
    return checkbox(options);
  }
}
```

```
    }  
  }  
  
  export const wrapText = (text, char = "=") => {  
    const rule = char.repeat(80);  
    return `${rule}\n  ${text}\n${rule}\n`;  
  };
```

Objects in S3 are stored in 'buckets'. Let's define a function for creating a new bucket.

```
export const createBucket = async () => {  
  const bucketName = await prompter.input({  
    message: "Enter a bucket name. Bucket names must be globally unique:",  
  });  
  const command = new CreateBucketCommand({ Bucket: bucketName });  
  await s3Client.send(command);  
  console.log("Bucket created successfully.\n");  
  return bucketName;  
};
```

Buckets contain 'objects'. This function uploads the contents of a directory to your bucket as objects.

```
export const uploadFilesToBucket = async ({ bucketName, folderPath }) => {  
  console.log(`Uploading files from ${folderPath}\n`);  
  const keys = readdirSync(folderPath);  
  const files = keys.map((key) => {  
    const filePath = `${folderPath}/${key}`;  
    const fileContent = readFileSync(filePath);  
    return {  
      Key: key,  
      Body: fileContent,  
    };  
  });  
  
  for (let file of files) {  
    await s3Client.send(  
      new PutObjectCommand({  
        Bucket: bucketName,  
        Body: file.Body,  
        Key: file.Key,  
      })  
    );  
  }  
};
```

```
    }),  
  );  
  console.log(`${file.Key} uploaded successfully.`);  
}  
};
```

After uploading objects, check to confirm that they were uploaded correctly. You can use `ListObjects` for that. You'll be using the 'Key' property, but there are other useful properties in the response also.

```
export const listFilesInBucket = async ({ bucketName }) => {  
  const command = new ListObjectsCommand({ Bucket: bucketName });  
  const { Contents } = await s3Client.send(command);  
  const contentsList = Contents.map((c) => ` • ${c.Key}`).join("\n");  
  console.log("\nHere's a list of files in the bucket:");  
  console.log(contentsList + "\n");  
};
```

Sometimes you might want to copy an object from one bucket to another. Use the `CopyObject` command for that.

```
export const copyFileFromBucket = async ({ destinationBucket }) => {  
  const proceed = await prompter.confirm({  
    message: "Would you like to copy an object from another bucket?",  
  });  
  
  if (!proceed) {  
    return;  
  } else {  
    const copy = async () => {  
      try {  
        const sourceBucket = await prompter.input({  
          message: "Enter source bucket name:",  
        });  
        const sourceKey = await prompter.input({  
          message: "Enter source key:",  
        });  
        const destinationKey = await prompter.input({  
          message: "Enter destination key:",  
        });  
      }  
    };  
  }  
};
```

```
    const command = new CopyObjectCommand({
      Bucket: destinationBucket,
      CopySource: `${sourceBucket}/${sourceKey}`,
      Key: destinationKey,
    });
    await s3Client.send(command);
    await copyFileFromBucket({ destinationBucket });
  } catch (err) {
    console.error(`Copy error.`);
    console.error(err);
    const retryAnswer = await prompter.confirm({ message: "Try again?" });
    if (retryAnswer) {
      await copy();
    }
  }
};
await copy();
}
```

There's no SDK method for getting multiple objects from a bucket. Instead, you'll create a list of objects to download and iterate over them.

```
export const downloadFilesFromBucket = async ({ bucketName }) => {
  const { Contents } = await s3Client.send(
    new ListObjectsCommand({ Bucket: bucketName }),
  );
  const path = await prompter.input({
    message: "Enter destination path for files:",
  });

  for (let content of Contents) {
    const obj = await s3Client.send(
      new GetObjectCommand({ Bucket: bucketName, Key: content.Key }),
    );
    writeFileSync(
      `${path}/${content.Key}`,
      await obj.Body.transformToByteArray(),
    );
  }
  console.log("Files downloaded successfully.\n");
}
```

```
};
```

It's time to clean up your resources. A bucket must be empty before it can be deleted. These two functions empty and delete the bucket.

```
export const emptyBucket = async ({ bucketName }) => {
  const listObjectsCommand = new ListObjectsCommand({ Bucket: bucketName });
  const { Contents } = await s3Client.send(listObjectsCommand);
  const keys = Contents.map((c) => c.Key);

  const deleteObjectsCommand = new DeleteObjectsCommand({
    Bucket: bucketName,
    Delete: { Objects: keys.map((key) => ({ Key: key })) },
  });
  await s3Client.send(deleteObjectsCommand);
  console.log(`${bucketName} emptied successfully.\n`);
};

export const deleteBucket = async ({ bucketName }) => {
  const command = new DeleteBucketCommand({ Bucket: bucketName });
  await s3Client.send(command);
  console.log(`${bucketName} deleted successfully.\n`);
};
```

The 'main' function pulls everything together. If you run this file directly the main function will be called.

```
const main = async () => {
  const OBJECT_DIRECTORY = `${dirnameFromMetaUrl(
    import.meta.url,
  )}../../../../resources/sample_files/.sample_media`;

  try {
    console.log(wrapText("Welcome to the Amazon S3 getting started example."));
    console.log("Let's create a bucket.");
    const bucketName = await createBucket();
    await prompter.confirm({ message: continueMessage });

    console.log(wrapText("File upload."));
    console.log(
```

```
    "I have some default files ready to go. You can edit the source code to
    provide your own.",
    );
    await uploadFilesToBucket({
      bucketName,
      folderPath: OBJECT_DIRECTORY,
    });

    await listFilesInBucket({ bucketName });
    await prompter.confirm({ message: continueMessage });

    console.log(wrapText("Copy files."));
    await copyFileFromBucket({ destinationBucket: bucketName });
    await listFilesInBucket({ bucketName });
    await prompter.confirm({ message: continueMessage });

    console.log(wrapText("Download files."));
    await downloadFilesFromBucket({ bucketName });

    console.log(wrapText("Clean up."));
    await emptyBucket({ bucketName });
    await deleteBucket({ bucketName });
  } catch (err) {
    console.error(err);
  }
};
```

- For API details, see the following topics in *AWS SDK for JavaScript API Reference*.
 - [CopyObject](#)
 - [CreateBucket](#)
 - [DeleteBucket](#)
 - [DeleteObjects](#)
 - [GetObject](#)
 - [ListObjectsV2](#)
 - [PutObject](#)

Actions

CopyObject

The following code example shows how to use CopyObject.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Copy the object.

```
import { S3Client, CopyObjectCommand } from "@aws-sdk/client-s3";

const client = new S3Client({});

export const main = async () => {
  const command = new CopyObjectCommand({
    CopySource: "SOURCE_BUCKET/SOURCE_OBJECT_KEY",
    Bucket: "DESTINATION_BUCKET",
    Key: "NEW_OBJECT_KEY",
  });

  try {
    const response = await client.send(command);
    console.log(response);
  } catch (err) {
    console.error(err);
  }
};
```

- For API details, see [CopyObject](#) in *AWS SDK for JavaScript API Reference*.

CreateBucket

The following code example shows how to use CreateBucket.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create the bucket.

```
import { CreateBucketCommand, S3Client } from "@aws-sdk/client-s3";

const client = new S3Client({});

export const main = async () => {
  const command = new CreateBucketCommand({
    // The name of the bucket. Bucket names are unique and have several other
    // constraints.
    // See https://docs.aws.amazon.com/AmazonS3/latest/userguide/
    // bucketnamingrules.html
    Bucket: "bucket-name",
  });

  try {
    const { Location } = await client.send(command);
    console.log(`Bucket created with location ${Location}`);
  } catch (err) {
    console.error(err);
  }
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [CreateBucket](#) in *AWS SDK for JavaScript API Reference*.

DeleteBucket

The following code example shows how to use DeleteBucket.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Delete the bucket.

```
import { DeleteBucketCommand, S3Client } from "@aws-sdk/client-s3";

const client = new S3Client({});

// Delete a bucket.
export const main = async () => {
  const command = new DeleteBucketCommand({
    Bucket: "test-bucket",
  });

  try {
    const response = await client.send(command);
    console.log(response);
  } catch (err) {
    console.error(err);
  }
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [DeleteBucket](#) in *AWS SDK for JavaScript API Reference*.

DeleteBucketPolicy

The following code example shows how to use DeleteBucketPolicy.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Delete the bucket policy.

```
import { DeleteBucketPolicyCommand, S3Client } from "@aws-sdk/client-s3";

const client = new S3Client({});

// This will remove the policy from the bucket.
export const main = async () => {
  const command = new DeleteBucketPolicyCommand({
    Bucket: "test-bucket",
  });

  try {
    const response = await client.send(command);
    console.log(response);
  } catch (err) {
    console.error(err);
  }
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [DeleteBucketPolicy](#) in *AWS SDK for JavaScript API Reference*.

DeleteBucketWebsite

The following code example shows how to use DeleteBucketWebsite.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Delete the website configuration from the bucket.

```
import { DeleteBucketWebsiteCommand, S3Client } from "@aws-sdk/client-s3";

const client = new S3Client({});

// Disable static website hosting on the bucket.
export const main = async () => {
  const command = new DeleteBucketWebsiteCommand({
    Bucket: "test-bucket",
  });

  try {
    const response = await client.send(command);
    console.log(response);
  } catch (err) {
    console.error(err);
  }
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [DeleteBucketWebsite](#) in *AWS SDK for JavaScript API Reference*.

DeleteObject

The following code example shows how to use DeleteObject.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Delete an object.

```
import { DeleteObjectCommand, S3Client } from "@aws-sdk/client-s3";

const client = new S3Client({});

export const main = async () => {
  const command = new DeleteObjectCommand({
    Bucket: "test-bucket",
    Key: "test-key.txt",
  });

  try {
    const response = await client.send(command);
    console.log(response);
  } catch (err) {
    console.error(err);
  }
};
```

- For API details, see [DeleteObject](#) in *AWS SDK for JavaScript API Reference*.

DeleteObjects

The following code example shows how to use DeleteObjects.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Delete multiple objects.

```
import { DeleteObjectsCommand, S3Client } from "@aws-sdk/client-s3";

const client = new S3Client({});

export const main = async () => {
  const command = new DeleteObjectsCommand({
    Bucket: "test-bucket",
    Delete: {
      Objects: [{ Key: "object1.txt" }, { Key: "object2.txt" }],
    },
  });

  try {
    const { Deleted } = await client.send(command);
    console.log(
      `Successfully deleted ${Deleted.length} objects from S3 bucket. Deleted
objects:`,
    );
    console.log(Deleted.map((d) => ` • ${d.Key}`).join("\n"));
  } catch (err) {
    console.error(err);
  }
};
```

- For API details, see [DeleteObjects](#) in *AWS SDK for JavaScript API Reference*.

GetBucketAcl

The following code example shows how to use `GetBucketAcl`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Get the ACL permissions.

```
import { GetBucketAclCommand, S3Client } from "@aws-sdk/client-s3";

const client = new S3Client({});

export const main = async () => {
  const command = new GetBucketAclCommand({
    Bucket: "test-bucket",
  });

  try {
    const response = await client.send(command);
    console.log(response);
  } catch (err) {
    console.error(err);
  }
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [GetBucketAcl](#) in *AWS SDK for JavaScript API Reference*.

GetBucketCors

The following code example shows how to use `GetBucketCors`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Get the CORS policy for the bucket.

```
import { GetBucketCorsCommand, S3Client } from "@aws-sdk/client-s3";

const client = new S3Client({});

export const main = async () => {
```

```
const command = new GetBucketCorsCommand({
  Bucket: "test-bucket",
});

try {
  const { CORSRules } = await client.send(command);
  CORSRules.forEach((cr, i) => {
    console.log(
      `\\nCORSRule ${i + 1}`,
      `\\n${"-".repeat(10)}`,
      `\\nAllowedHeaders: ${cr.AllowedHeaders.join(" ")}`,
      `\\nAllowedMethods: ${cr.AllowedMethods.join(" ")}`,
      `\\nAllowedOrigins: ${cr.AllowedOrigins.join(" ")}`,
      `\\nExposeHeaders: ${cr.ExposeHeaders.join(" ")}`,
      `\\nMaxAgeSeconds: ${cr.MaxAgeSeconds}`,
    );
  });
} catch (err) {
  console.error(err);
}
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [GetBucketCors](#) in *AWS SDK for JavaScript API Reference*.

GetBucketPolicy

The following code example shows how to use `GetBucketPolicy`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Get the bucket policy.

```
import { GetBucketPolicyCommand, S3Client } from "@aws-sdk/client-s3";
```

```
const client = new S3Client({});

export const main = async () => {
  const command = new GetBucketPolicyCommand({
    Bucket: "test-bucket",
  });

  try {
    const { Policy } = await client.send(command);
    console.log(JSON.parse(Policy));
  } catch (err) {
    console.error(err);
  }
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [GetBucketPolicy](#) in *AWS SDK for JavaScript API Reference*.

GetBucketWebsite

The following code example shows how to use GetBucketWebsite.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Get the website configuration.

```
import { GetBucketWebsiteCommand, S3Client } from "@aws-sdk/client-s3";

const client = new S3Client({});

export const main = async () => {
  const command = new GetBucketWebsiteCommand({
    Bucket: "test-bucket",
```



```
});

try {
  const { ErrorDocument, IndexDocument } = await client.send(command);
  console.log(
    `Your bucket is set up to host a website. It has an error document:`,
    `${ErrorDocument.Key}, and an index document: ${IndexDocument.Suffix}.`,
  );
} catch (err) {
  console.error(err);
}
};
```

- For API details, see [GetBucketWebsite](#) in *AWS SDK for JavaScript API Reference*.

GetObject

The following code example shows how to use `GetObject`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Download the object.

```
import { GetObjectCommand, S3Client } from "@aws-sdk/client-s3";

const client = new S3Client({});

export const main = async () => {
  const command = new GetObjectCommand({
    Bucket: "test-bucket",
    Key: "hello-s3.txt",
  });

  try {
    const response = await client.send(command);
```

```
// The Body object also has 'transformToByteArray' and 'transformToWebStream'
methods.
const str = await response.Body.transformToString();
console.log(str);
} catch (err) {
  console.error(err);
}
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [GetObject](#) in *AWS SDK for JavaScript API Reference*.

GetObjectLockConfiguration

The following code example shows how to use GetObjectLockConfiguration.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { fileURLToPath } from "url";
import {
  GetObjectLockConfigurationCommand,
  S3Client,
} from "@aws-sdk/client-s3";

/**
 * @param {S3Client} client
 * @param {string} bucketName
 */
export const main = async (client, bucketName) => {
  const command = new GetObjectLockConfigurationCommand({
    Bucket: bucketName,
    // Optionally, you can provide additional parameters
    // ExpectedBucketOwner: "ACCOUNT_ID",
  });
```

```
try {
  const { ObjectLockConfiguration } = await client.send(command);
  console.log(`Object Lock Configuration: ${ObjectLockConfiguration}`);
} catch (err) {
  console.error(err);
}
};

// Invoke main function if this file was run directly.
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  main(new S3Client(), "BUCKET_NAME");
}
```

- For API details, see [GetObjectLockConfiguration](#) in *AWS SDK for JavaScript API Reference*.

GetObjectRetention

The following code example shows how to use `GetObjectRetention`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { fileURLToPath } from "url";
import { GetObjectRetentionCommand, S3Client } from "@aws-sdk/client-s3";

/**
 * @param {S3Client} client
 * @param {string} bucketName
 * @param {string} objectKey
 */
export const main = async (client, bucketName, objectKey) => {
  const command = new GetObjectRetentionCommand({
    Bucket: bucketName,
    Key: objectKey,
```

```
// Optionally, you can provide additional parameters
// ExpectedBucketOwner: "ACCOUNT_ID",
// RequestPayer: "requester",
// VersionId: "OBJECT_VERSION_ID",
});

try {
  const { Retention } = await client.send(command);
  console.log(`Object Retention Settings: ${Retention.Status}`);
} catch (err) {
  console.error(err);
}
};

// Invoke main function if this file was run directly.
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  main(new S3Client(), "BUCKET_NAME", "OBJECT_KEY");
}
```

- For API details, see [GetObjectRetention](#) in *AWS SDK for JavaScript API Reference*.

ListBuckets

The following code example shows how to use ListBuckets.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

List the buckets.

```
import { ListBucketsCommand, S3Client } from "@aws-sdk/client-s3";

const client = new S3Client({});

export const main = async () => {
  const command = new ListBucketsCommand({});
```

```
try {
  const { Owner, Buckets } = await client.send(command);
  console.log(
    `${Owner.DisplayName} owns ${Buckets.length} bucket${
      Buckets.length === 1 ? "" : "s"
    }:` ,
  );
  console.log(`${Buckets.map((b) => ` • ${b.Name}`)}`.join("\n"));
} catch (err) {
  console.error(err);
}
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [ListBuckets](#) in *AWS SDK for JavaScript API Reference*.

ListObjectsV2

The following code example shows how to use ListObjectsV2.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

List all of the objects in your bucket. If there is more than one object, `IsTruncated` and `NextContinuationToken` will be used to iterate over the full list.

```
import {
  S3Client,
  // This command supersedes the ListObjectsCommand and is the recommended way to
  list objects.
  ListObjectsV2Command,
} from "@aws-sdk/client-s3";

const client = new S3Client({});
```

```
export const main = async () => {
  const command = new ListObjectsV2Command({
    Bucket: "my-bucket",
    // The default and maximum number of keys returned is 1000. This limits it to
    // one for demonstration purposes.
    MaxKeys: 1,
  });

  try {
    let isTruncated = true;

    console.log("Your bucket contains the following objects:\n");
    let contents = "";

    while (isTruncated) {
      const { Contents, IsTruncated, NextContinuationToken } =
        await client.send(command);
      const contentsList = Contents.map((c) => ` • ${c.Key}`).join("\n");
      contents += contentsList + "\n";
      isTruncated = IsTruncated;
      command.input.ContinuationToken = NextContinuationToken;
    }
    console.log(contents);
  } catch (err) {
    console.error(err);
  }
};
```

- For API details, see [ListObjectsV2](#) in *AWS SDK for JavaScript API Reference*.

PutBucketAcl

The following code example shows how to use PutBucketAcl.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Put the bucket ACL.

```
import { PutBucketAclCommand, S3Client } from "@aws-sdk/client-s3";

const client = new S3Client({});

// Most Amazon S3 use cases don't require the use of access control lists (ACLs).
// We recommend that you disable ACLs, except in unusual circumstances where
// you need to control access for each object individually.
// Consider a policy instead. For more information see https://docs.aws.amazon.com/AmazonS3/latest/userguide/bucket-policies.html.
export const main = async () => {
  // Grant a user READ access to a bucket.
  const command = new PutBucketAclCommand({
    Bucket: "test-bucket",
    AccessControlPolicy: {
      Grants: [
        {
          Grantee: {
            // The canonical ID of the user. This ID is an obfuscated form of your
            // AWS account number.
            // It's unique to Amazon S3 and can't be found elsewhere.
            // For more information, see https://docs.aws.amazon.com/AmazonS3/latest/userguide/finding-canonical-user-id.html.
            ID: "canonical-id-1",
            Type: "CanonicalUser",
          },
          // One of FULL_CONTROL | READ | WRITE | READ_ACP | WRITE_ACP
          // https://docs.aws.amazon.com/AmazonS3/latest/API/API\_Grant.html#AmazonS3-Type-Grant-Permission
          Permission: "FULL_CONTROL",
        },
      ],
      Owner: {
        ID: "canonical-id-2",
      },
    },
  });

  try {
    const response = await client.send(command);
    console.log(response);
  } catch (err) {
    console.error(err);
  }
}
```

```
  }  
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [PutBucketAcl](#) in *AWS SDK for JavaScript API Reference*.

PutBucketCors

The following code example shows how to use PutBucketCors.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Add a CORS rule.

```
import { PutBucketCorsCommand, S3Client } from "@aws-sdk/client-s3";  
  
const client = new S3Client({});  
  
// By default, Amazon S3 doesn't allow cross-origin requests. Use this command  
// to explicitly allow cross-origin requests.  
export const main = async () => {  
  const command = new PutBucketCorsCommand({  
    Bucket: "test-bucket",  
    CORSConfiguration: {  
      CORSRules: [  
        {  
          // Allow all headers to be sent to this bucket.  
          AllowedHeaders: ["*"],  
          // Allow only GET and PUT methods to be sent to this bucket.  
          AllowedMethods: ["GET", "PUT"],  
          // Allow only requests from the specified origin.  
          AllowedOrigins: ["https://www.example.com"],  
          // Allow the entity tag (ETag) header to be returned in the response. The  
          ETag header
```



```
        // The entity tag represents a specific version of the object. The ETag
reflects
        // changes only to the contents of an object, not its metadata.
        ExposeHeaders: ["ETag"],
        // How long the requesting browser should cache the preflight response.
After
        // this time, the preflight request will have to be made again.
        MaxAgeSeconds: 3600,
    },
],
},
});

try {
    const response = await client.send(command);
    console.log(response);
} catch (err) {
    console.error(err);
}
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [PutBucketCors](#) in *AWS SDK for JavaScript API Reference*.

PutBucketPolicy

The following code example shows how to use PutBucketPolicy.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Add the policy.

```
import { PutBucketPolicyCommand, S3Client } from "@aws-sdk/client-s3";
```

```
const client = new S3Client({});

export const main = async () => {
  const command = new PutBucketPolicyCommand({
    Policy: JSON.stringify({
      Version: "2012-10-17",
      Statement: [
        {
          Sid: "AllowGetObject",
          // Allow this particular user to call GetObject on any object in this
bucket.
          Effect: "Allow",
          Principal: {
            AWS: "arn:aws:iam::ACCOUNT-ID:user/USERNAME",
          },
          Action: "s3:GetObject",
          Resource: "arn:aws:s3:::BUCKET-NAME/*",
        },
      ],
    }),
    // Apply the preceding policy to this bucket.
    Bucket: "BUCKET-NAME",
  });

  try {
    const response = await client.send(command);
    console.log(response);
  } catch (err) {
    console.error(err);
  }
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [PutBucketPolicy](#) in *AWS SDK for JavaScript API Reference*.

PutBucketWebsite

The following code example shows how to use PutBucketWebsite.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Set the website configuration.

```
import { PutBucketWebsiteCommand, S3Client } from "@aws-sdk/client-s3";

const client = new S3Client({});

// Set up a bucket as a static website.
// The bucket needs to be publicly accessible.
export const main = async () => {
  const command = new PutBucketWebsiteCommand({
    Bucket: "test-bucket",
    WebsiteConfiguration: {
      ErrorDocument: {
        // The object key name to use when a 4XX class error occurs.
        Key: "error.html",
      },
      IndexDocument: {
        // A suffix that is appended to a request that is for a directory.
        Suffix: "index.html",
      },
    },
  });

  try {
    const response = await client.send(command);
    console.log(response);
  } catch (err) {
    console.error(err);
  }
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [PutBucketWebsite](#) in *AWS SDK for JavaScript API Reference*.

PutObject

The following code example shows how to use PutObject.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Upload the object.

```
import { PutObjectCommand, S3Client } from "@aws-sdk/client-s3";

const client = new S3Client({});

export const main = async () => {
  const command = new PutObjectCommand({
    Bucket: "test-bucket",
    Key: "hello-s3.txt",
    Body: "Hello S3!",
  });

  try {
    const response = await client.send(command);
    console.log(response);
  } catch (err) {
    console.error(err);
  }
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [PutObject](#) in *AWS SDK for JavaScript API Reference*.

PutObjectLegalHold

The following code example shows how to use PutObjectLegalHold.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { fileURLToPath } from "url";
import { PutObjectLegalHoldCommand, S3Client } from "@aws-sdk/client-s3";

/**
 * @param {S3Client} client
 * @param {string} bucketName
 * @param {string} objectKey
 */
export const main = async (client, bucketName, objectKey) => {
  const command = new PutObjectLegalHoldCommand({
    Bucket: bucketName,
    Key: objectKey,
    LegalHold: {
      // Set the status to 'ON' to place a legal hold on the object.
      // Set the status to 'OFF' to remove the legal hold.
      Status: "ON",
    },
    // Optionally, you can provide additional parameters
    // ChecksumAlgorithm: "ALGORITHM",
    // ContentMD5: "MD5_HASH",
    // ExpectedBucketOwner: "ACCOUNT_ID",
    // RequestPayer: "requester",
    // VersionId: "OBJECT_VERSION_ID",
  });

  try {
    const response = await client.send(command);
    console.log(
      `Object legal hold status: ${response.$metadata.httpStatusCode}`
    );
  } catch (err) {
    console.error(err);
  }
};
```

```
// Invoke main function if this file was run directly.
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  main(new S3Client(), "BUCKET_NAME", "OBJECT_KEY");
}
```

- For API details, see [PutObjectLegalHold](#) in *AWS SDK for JavaScript API Reference*.

PutObjectLockConfiguration

The following code example shows how to use PutObjectLockConfiguration.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Set the object lock configuration of a bucket.

```
import { fileURLToPath } from "url";
import {
  PutObjectLockConfigurationCommand,
  S3Client,
} from "@aws-sdk/client-s3";

/**
 * @param {S3Client} client
 * @param {string} bucketName
 */
export const main = async (client, bucketName) => {
  const command = new PutObjectLockConfigurationCommand({
    Bucket: bucketName,
    // The Object Lock configuration that you want to apply to the specified bucket.
    ObjectLockConfiguration: {
      ObjectLockEnabled: "Enabled",
    },
    // Optionally, you can provide additional parameters
    // ExpectedBucketOwner: "ACCOUNT_ID",
  });
```

```

    // RequestPayer: "requester",
    // Token: "OPTIONAL_TOKEN",
  });

  try {
    const response = await client.send(command);
    console.log(
      `Object Lock Configuration updated: ${response.$metadata.httpStatusCode}`,
    );
  } catch (err) {
    console.error(err);
  }
};

// Invoke main function if this file was run directly.
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  main(new S3Client(), "BUCKET_NAME");
}

```

Set the default retention period of a bucket.

```

import { fileURLToPath } from "url";
import {
  PutObjectLockConfigurationCommand,
  S3Client,
} from "@aws-sdk/client-s3";

/**
 * @param {S3Client} client
 * @param {string} bucketName
 */
export const main = async (client, bucketName) => {
  const command = new PutObjectLockConfigurationCommand({
    Bucket: bucketName,
    // The Object Lock configuration that you want to apply to the specified bucket.
    ObjectLockConfiguration: {
      ObjectLockEnabled: "Enabled",
      Rule: {
        DefaultRetention: {
          Mode: "GOVERNANCE",
          Years: 3,
        },
      },
    },
  });

```

```

    },
  },
  // Optionally, you can provide additional parameters
  // ExpectedBucketOwner: "ACCOUNT_ID",
  // RequestPayer: "requester",
  // Token: "OPTIONAL_TOKEN",
});

try {
  const response = await client.send(command);
  console.log(
    `Default Object Lock Configuration updated: ${response.
$metadata.httpStatusCode}`,
  );
} catch (err) {
  console.error(err);
}
};

// Invoke main function if this file was run directly.
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  main(new S3Client(), "BUCKET_NAME");
}

```

- For API details, see [PutObjectLockConfiguration](#) in *AWS SDK for JavaScript API Reference*.

PutObjectRetention

The following code example shows how to use PutObjectRetention.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```

import { fileURLToPath } from "url";
import { PutObjectRetentionCommand, S3Client } from "@aws-sdk/client-s3";

```



```
/**
 * @param {S3Client} client
 * @param {string} bucketName
 * @param {string} objectKey
 */
export const main = async (client, bucketName, objectKey) => {
  const command = new PutObjectRetentionCommand({
    Bucket: bucketName,
    Key: objectKey,
    BypassGovernanceRetention: false,
    // ChecksumAlgorithm: "ALGORITHM",
    // ContentMD5: "MD5_HASH",
    // ExpectedBucketOwner: "ACCOUNT_ID",
    // RequestPayer: "requester",
    Retention: {
      Mode: "GOVERNANCE", // or "COMPLIANCE"
      RetainUntilDate: new Date(new Date().getTime() + 24 * 60 * 60 * 1000),
    },
    // VersionId: "OBJECT_VERSION_ID",
  });

  try {
    const response = await client.send(command);
    console.log(
      `Object Retention settings updated: ${response.$metadata.httpStatusCode}`,
    );
  } catch (err) {
    console.error(err);
  }
};

// Invoke main function if this file was run directly.
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  main(new S3Client(), "BUCKET_NAME", "OBJECT_KEY");
}
```

- For API details, see [PutObjectRetention](#) in *AWS SDK for JavaScript API Reference*.

Scenarios

Create a presigned URL

The following code example shows how to create a presigned URL for Amazon S3 and upload an object.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create a presigned URL to upload an object to a bucket.

```
import https from "https";
import { PutObjectCommand, S3Client } from "@aws-sdk/client-s3";
import { fromIni } from "@aws-sdk/credential-providers";
import { HttpRequest } from "@smithy/protocol-http";
import {
  getSignedUrl,
  S3RequestPresigner,
} from "@aws-sdk/s3-request-presigner";
import { parseUrl } from "@smithy/url-parser";
import { formatUrl } from "@aws-sdk/util-format-url";
import { Hash } from "@smithy/hash-node";

const createPresignedUrlWithoutClient = async ({ region, bucket, key }) => {
  const url = parseUrl(`https://${bucket}.s3.${region}.amazonaws.com/${key}`);
  const presigner = new S3RequestPresigner({
    credentials: fromIni(),
    region,
    sha256: Hash.bind(null, "sha256"),
  });

  const signedUrlObject = await presigner.presign(
    new HttpRequest({ ...url, method: "PUT" }),
  );
  return formatUrl(signedUrlObject);
};
```

```
const createPresignedUrlWithClient = ({ region, bucket, key }) => {
  const client = new S3Client({ region });
  const command = new PutObjectCommand({ Bucket: bucket, Key: key });
  return getSignedUrl(client, command, { expiresIn: 3600 });
};

function put(url, data) {
  return new Promise((resolve, reject) => {
    const req = https.request(
      url,
      { method: "PUT", headers: { "Content-Length": new Blob([data]).size } },
      (res) => {
        let responseBody = "";
        res.on("data", (chunk) => {
          responseBody += chunk;
        });
        res.on("end", () => {
          resolve(responseBody);
        });
      },
    );
    req.on("error", (err) => {
      reject(err);
    });
    req.write(data);
    req.end();
  });
}

export const main = async () => {
  const REGION = "us-east-1";
  const BUCKET = "example_bucket";
  const KEY = "example_file.txt";

  // There are two ways to generate a presigned URL.
  // 1. Use createPresignedUrl without the S3 client.
  // 2. Use getSignedUrl in conjunction with the S3 client and GetObjectCommand.
  try {
    const noClientUrl = await createPresignedUrlWithoutClient({
      region: REGION,
      bucket: BUCKET,
      key: KEY,
    });
  }
};
```

```
const clientUrl = await createPresignedUrlWithClient({
  region: REGION,
  bucket: BUCKET,
  key: KEY,
});

// After you get the presigned URL, you can provide your own file
// data. Refer to put() above.
console.log("Calling PUT using presigned URL without client");
await put(noClientUrl, "Hello World");

console.log("Calling PUT using presigned URL with client");
await put(clientUrl, "Hello World");

console.log("\nDone. Check your S3 console.");
} catch (err) {
  console.error(err);
}
};
```

Create a presigned URL to download an object from a bucket.

```
import { GetObjectCommand, S3Client } from "@aws-sdk/client-s3";
import { fromIni } from "@aws-sdk/credential-providers";
import { HttpRequest } from "@smithy/protocol-http";
import {
  getSignedUrl,
  S3RequestPresigner,
} from "@aws-sdk/s3-request-presigner";
import { parseUrl } from "@smithy/url-parser";
import { formatUrl } from "@aws-sdk/util-format-url";
import { Hash } from "@smithy/hash-node";

const createPresignedUrlWithoutClient = async ({ region, bucket, key }) => {
  const url = parseUrl(`https://${bucket}.s3.${region}.amazonaws.com/${key}`);
  const presigner = new S3RequestPresigner({
    credentials: fromIni(),
    region,
    sha256: Hash.bind(null, "sha256"),
  });
```

```
const signedUrlObject = await presigner.presign(new HttpRequest(url));
return formatUrl(signedUrlObject);
};

const createPresignedUrlWithClient = ({ region, bucket, key }) => {
  const client = new S3Client({ region });
  const command = new GetObjectCommand({ Bucket: bucket, Key: key });
  return getSignedUrl(client, command, { expiresIn: 3600 });
};

export const main = async () => {
  const REGION = "us-east-1";
  const BUCKET = "example_bucket";
  const KEY = "example_file.jpg";

  try {
    const noClientUrl = await createPresignedUrlWithoutClient({
      region: REGION,
      bucket: BUCKET,
      key: KEY,
    });

    const clientUrl = await createPresignedUrlWithClient({
      region: REGION,
      bucket: BUCKET,
      key: KEY,
    });

    console.log("Presigned URL without client");
    console.log(noClientUrl);
    console.log("\n");

    console.log("Presigned URL with client");
    console.log(clientUrl);
  } catch (err) {
    console.error(err);
  }
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).

Create a serverless application to manage photos

The following code example shows how to create a serverless application that lets users manage photos using labels.

SDK for JavaScript (v3)

Shows how to develop a photo asset management application that detects labels in images using Amazon Rekognition and stores them for later retrieval.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

For a deep dive into the origin of this example see the post on [AWS Community](#).

Services used in this example

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Create a web page that lists Amazon S3 objects

The following code example shows how to list Amazon S3 objects in a web page.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

The following code is the relevant React component that makes calls to the AWS SDK. A runnable version of the application containing this component can be found at the preceding GitHub link.

```
import { useEffect, useState } from "react";
import {
  ListObjectsCommand,
  ListObjectsCommandOutput,
  S3Client,
} from "@aws-sdk/client-s3";
import { fromCognitoIdentityPool } from "@aws-sdk/credential-providers";
import "./App.css";

function App() {
  const [objects, setObjects] = useState<
    Required<ListObjectsCommandOutput>["Contents"]
  >([]);

  useEffect(() => {
    const client = new S3Client({
      region: "us-east-1",
      // Unless you have a public bucket, you'll need access to a private bucket.
      // One way to do this is to create an Amazon Cognito identity pool, attach a
role to the pool,
      // and grant the role access to the 's3:GetObject' action.
      //
      // You'll also need to configure the CORS settings on the bucket to allow
traffic from
      // this example site. Here's an example configuration that allows all origins.
Don't
      // do this in production.
      // [
      // {
      //   "AllowedHeaders": ["*"],
      //   "AllowedMethods": ["GET"],
      //   "AllowedOrigins": ["*"],
      //   "ExposeHeaders": [],
      // },
      // ]
      //
      credentials: fromCognitoIdentityPool({
        clientConfig: { region: "us-east-1" },
        identityPoolId: "<YOUR_IDENTITY_POOL_ID>",
      }),
    });
    const command = new ListObjectsCommand({ Bucket: "bucket-name" });
    client.send(command).then(({ Contents }) => setObjects(Contents || []));
  });
}
```

```
    }, []);

    return (
      <div className="App">
        {objects.map((o) => (
          <div key={o.ETag}>{o.Key}</div>
        ))}
      </div>
    );
  }
}

export default App;
```

- For API details, see [ListObjects](#) in *AWS SDK for JavaScript API Reference*.

Create an Amazon Textract explorer application

The following code example shows how to explore Amazon Textract output through an interactive application.

SDK for JavaScript (v3)

Shows how to use the AWS SDK for JavaScript to build a React application that uses Amazon Textract to extract data from a document image and display it in an interactive web page. This example runs in a web browser and requires an authenticated Amazon Cognito identity for credentials. It uses Amazon Simple Storage Service (Amazon S3) for storage, and for notifications it polls an Amazon Simple Queue Service (Amazon SQS) queue that is subscribed to an Amazon Simple Notification Service (Amazon SNS) topic.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- Amazon Cognito Identity
- Amazon S3
- Amazon SNS
- Amazon SQS
- Amazon Textract

Detect PPE in images

The following code example shows how to build an app that uses Amazon Rekognition to detect Personal Protective Equipment (PPE) in images.

SDK for JavaScript (v3)

Shows how to use Amazon Rekognition with the AWS SDK for JavaScript to create an application to detect personal protective equipment (PPE) in images located in an Amazon Simple Storage Service (Amazon S3) bucket. The app saves the results to an Amazon DynamoDB table, and sends the admin an email notification with the results using Amazon Simple Email Service (Amazon SES).

Learn how to:

- Create an unauthenticated user using Amazon Cognito.
- Analyze images for PPE using Amazon Rekognition.
- Verify an email address for Amazon SES.
- Update a DynamoDB table with results.
- Send an email notification using Amazon SES.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- DynamoDB
- Amazon Rekognition
- Amazon S3
- Amazon SES

Detect objects in images

The following code example shows how to build an app that uses Amazon Rekognition to detect objects by category in images.

SDK for JavaScript (v3)

Shows how to use Amazon Rekognition with the AWS SDK for JavaScript to create an app that uses Amazon Rekognition to identify objects by category in images located in an Amazon

Simple Storage Service (Amazon S3) bucket. The app sends the admin an email notification with the results using Amazon Simple Email Service (Amazon SES).

Learn how to:

- Create an unauthenticated user using Amazon Cognito.
- Analyze images for objects using Amazon Rekognition.
- Verify an email address for Amazon SES.
- Send an email notification using Amazon SES.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- Amazon Rekognition
- Amazon S3
- Amazon SES

Get the legal hold configuration of an object

The following code example shows how to get the legal hold configuration of an S3 bucket.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { fileURLToPath } from "url";
import { GetObjectLegalHoldCommand, S3Client } from "@aws-sdk/client-s3";

/**
 * @param {S3Client} client
 * @param {string} bucketName
 * @param {string} objectKey
 */
export const main = async (client, bucketName, objectKey) => {
```

```
const command = new GetObjectLegalHoldCommand({
  Bucket: bucketName,
  Key: objectKey,
  // Optionally, you can provide additional parameters
  // ExpectedBucketOwner: "ACCOUNT_ID",
  // RequestPayer: "requester",
  // VersionId: "OBJECT_VERSION_ID",
});

try {
  const response = await client.send(command);
  console.log(`Legal Hold Status: ${response.LegalHold.Status}`);
} catch (err) {
  console.error(err);
}

// Invoke main function if this file was run directly.
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  main(new S3Client(), "DOC-EXAMPLE-BUCKET", "OBJECT_KEY");
}
```

- For API details, see [GetObjectLegalHold](#) in *AWS SDK for JavaScript API Reference*.

Lock Amazon S3 objects

The following code example shows how to work with S3 object lock features.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

index.js - Entrypoint for the workflow. This orchestrates all of the steps. Visit GitHub to see the implementation details for Scenario, ScenarioInput, ScenarioOutput, and ScenarioAction.

```
import * as Scenarios from "@aws-doc-sdk-examples/lib/scenario/index.js";
import {
```

```
    exitOnFalse,
    loadState,
    saveState,
  } from "@aws-doc-sdk-examples/lib/scenario/steps-common.js";

import { welcome, welcomeContinue } from "../welcome.steps.js";
import {
  confirmCreateBuckets,
  confirmPopulateBuckets,
  confirmSetLegalHoldFileEnabled,
  confirmSetLegalHoldFileRetention,
  confirmSetRetentionPeriodFileEnabled,
  confirmSetRetentionPeriodFileRetention,
  confirmUpdateLockPolicy,
  confirmUpdateRetention,
  createBuckets,
  createBucketsAction,
  populateBuckets,
  populateBucketsAction,
  setLegalHoldFileEnabledAction,
  setLegalHoldFileRetentionAction,
  setRetentionPeriodFileEnabledAction,
  setRetentionPeriodFileRetentionAction,
  updateLockPolicy,
  updateLockPolicyAction,
  updateRetention,
  updateRetentionAction,
} from "../setup.steps.js";

/**
 * @param {Scenarios} scenarios
 * @param {Record<string, any>} initialState
 */
export const getWorkflowStages = (scenarios, initialState = {}) => {
  const client = new S3Client({});

  return {
    deploy: new scenarios.Scenario(
      "S3 Object Locking - Deploy",
      [
        welcome(scenarios),
        welcomeContinue(scenarios),
        exitOnFalse(scenarios, "welcomeContinue"),
        createBuckets(scenarios),
```

```
        confirmCreateBuckets(scenarios),
        exitOnFalse(scenarios, "confirmCreateBuckets"),
        createBucketsAction(scenarios, client),
        updateRetention(scenarios),
        confirmUpdateRetention(scenarios),
        exitOnFalse(scenarios, "confirmUpdateRetention"),
        updateRetentionAction(scenarios, client),
        populateBuckets(scenarios),
        confirmPopulateBuckets(scenarios),
        exitOnFalse(scenarios, "confirmPopulateBuckets"),
        populateBucketsAction(scenarios, client),
        updateLockPolicy(scenarios),
        confirmUpdateLockPolicy(scenarios),
        exitOnFalse(scenarios, "confirmUpdateLockPolicy"),
        updateLockPolicyAction(scenarios, client),
        confirmSetLegalHoldFileEnabled(scenarios),
        setLegalHoldFileEnabledAction(scenarios, client),
        confirmSetRetentionPeriodFileEnabled(scenarios),
        setRetentionPeriodFileEnabledAction(scenarios, client),
        confirmSetLegalHoldFileRetention(scenarios),
        setLegalHoldFileRetentionAction(scenarios, client),
        confirmSetRetentionPeriodFileRetention(scenarios),
        setRetentionPeriodFileRetentionAction(scenarios, client),
        saveState,
    ],
    initialState,
),
demo: new scenarios.Scenario(
    "S3 Object Locking - Demo",
    [loadState, replAction(scenarios, client)],
    initialState,
),
clean: new scenarios.Scenario(
    "S3 Object Locking - Destroy",
    [
        loadState,
        confirmCleanup(scenarios),
        exitOnFalse(scenarios, "confirmCleanup"),
        cleanupAction(scenarios, client),
    ],
    initialState,
),
};
};
```

```
// Call function if run directly
import { fileURLToPath } from "url";
import { S3Client } from "@aws-sdk/client-s3";
import { cleanupAction, confirmCleanup } from "./clean.steps.js";
import { replAction } from "./repl.steps.js";

if (process.argv[1] === fileURLToPath(import.meta.url)) {
  const objectLockingScenarios = getWorkflowStages(Scenarios);
  Scenarios.parseScenarioArgs(objectLockingScenarios);
}
```

welcome.steps.js - Output welcome messages to the console.

```
/**
 * @typedef {import("@aws-doc-sdk-examples/lib/scenario/index.js")} Scenarios
 */

/**
 * @param {Scenarios} scenarios
 */
const welcome = (scenarios) =>
  new scenarios.ScenarioOutput(
    "welcome",
    `Welcome to the Amazon Simple Storage Service (S3) Object Locking Workflow
    Scenario. For this workflow, we will use the AWS SDK for JavaScript to create
    several S3 buckets and files to demonstrate working with S3 locking features.`,
    { header: true },
  );

/**
 * @param {Scenarios} scenarios
 */
const welcomeContinue = (scenarios) =>
  new scenarios.ScenarioInput(
    "welcomeContinue",
    "Press Enter when you are ready to start.",
    { type: "confirm" },
  );

export { welcome, welcomeContinue };
```

setup.steps.js - Deploy buckets, objects, and file settings.

```
import {
  BucketVersioningStatus,
  ChecksumAlgorithm,
  CreateBucketCommand,
  MFADeleteStatus,
  PutBucketVersioningCommand,
  PutObjectCommand,
  PutObjectLockConfigurationCommand,
  PutObjectLegalHoldCommand,
  PutObjectRetentionCommand,
  ObjectLockLegalHoldStatus,
  ObjectLockRetentionMode,
} from "@aws-sdk/client-s3";

/**
 * @typedef {import("@aws-doc-sdk-examples/lib/scenario/index.js")} Scenarios
 */

/**
 * @typedef {import("@aws-sdk/client-s3").S3Client} S3Client
 */

const bucketPrefix = "js-object-locking";

/**
 * @param {Scenarios} scenarios
 * @param {S3Client} client
 */
const createBuckets = (scenarios) =>
  new scenarios.ScenarioOutput(
    "createBuckets",
    `The following buckets will be created:
      ${bucketPrefix}-no-lock with object lock False.
      ${bucketPrefix}-lock-enabled with object lock True.
      ${bucketPrefix}-retention-after-creation with object lock False.`,
    { preformatted: true },
  );

/**
 * @param {Scenarios} scenarios
 */
const confirmCreateBuckets = (scenarios) =>
```

```
new scenarios.ScenarioInput("confirmCreateBuckets", "Create the buckets?", {
  type: "confirm",
});

/**
 * @param {Scenarios} scenarios
 * @param {S3Client} client
 */
const createBucketsAction = (scenarios, client) =>
  new scenarios.ScenarioAction("createBucketsAction", async (state) => {
    const noLockBucketName = `${bucketPrefix}-no-lock`;
    const lockEnabledBucketName = `${bucketPrefix}-lock-enabled`;
    const retentionBucketName = `${bucketPrefix}-retention-after-creation`;

    await client.send(new CreateBucketCommand({ Bucket: noLockBucketName }));
    await client.send(
      new CreateBucketCommand({
        Bucket: lockEnabledBucketName,
        ObjectLockEnabledForBucket: true,
      }),
    );
    await client.send(new CreateBucketCommand({ Bucket: retentionBucketName }));

    state.noLockBucketName = noLockBucketName;
    state.lockEnabledBucketName = lockEnabledBucketName;
    state.retentionBucketName = retentionBucketName;
  });

/**
 * @param {Scenarios} scenarios
 */
const populateBuckets = (scenarios) =>
  new scenarios.ScenarioOutput(
    "populateBuckets",
    `The following test files will be created:
      file0.txt in ${bucketPrefix}-no-lock.
      file1.txt in ${bucketPrefix}-no-lock.
      file0.txt in ${bucketPrefix}-lock-enabled.
      file1.txt in ${bucketPrefix}-lock-enabled.
      file0.txt in ${bucketPrefix}-retention-after-creation.
      file1.txt in ${bucketPrefix}-retention-after-creation.`
    ,
    { preformatted: true },
  );
```



```
/**
 * @param {Scenarios} scenarios
 */
const confirmPopulateBuckets = (scenarios) =>
  new scenarios.ScenarioInput(
    "confirmPopulateBuckets",
    "Populate the buckets?",
    { type: "confirm" },
  );

/**
 * @param {Scenarios} scenarios
 * @param {S3Client} client
 */
const populateBucketsAction = (scenarios, client) =>
  new scenarios.ScenarioAction("populateBucketsAction", async (state) => {
    await client.send(
      new PutObjectCommand({
        Bucket: state.noLockBucketName,
        Key: "file0.txt",
        Body: "Content",
        ChecksumAlgorithm: ChecksumAlgorithm.SHA256,
      }),
    );
    await client.send(
      new PutObjectCommand({
        Bucket: state.noLockBucketName,
        Key: "file1.txt",
        Body: "Content",
        ChecksumAlgorithm: ChecksumAlgorithm.SHA256,
      }),
    );
    await client.send(
      new PutObjectCommand({
        Bucket: state.lockEnabledBucketName,
        Key: "file0.txt",
        Body: "Content",
        ChecksumAlgorithm: ChecksumAlgorithm.SHA256,
      }),
    );
    await client.send(
      new PutObjectCommand({
        Bucket: state.lockEnabledBucketName,
        Key: "file1.txt",
```

```
        Body: "Content",
        ChecksumAlgorithm: ChecksumAlgorithm.SHA256,
    })),
);
await client.send(
    new PutObjectCommand({
        Bucket: state.retentionBucketName,
        Key: "file0.txt",
        Body: "Content",
        ChecksumAlgorithm: ChecksumAlgorithm.SHA256,
    })),
);
await client.send(
    new PutObjectCommand({
        Bucket: state.retentionBucketName,
        Key: "file1.txt",
        Body: "Content",
        ChecksumAlgorithm: ChecksumAlgorithm.SHA256,
    })),
);
});

/**
 * @param {Scenarios} scenarios
 */
const updateRetention = (scenarios) =>
    new scenarios.ScenarioOutput(
        "updateRetention",
        `A bucket can be configured to use object locking with a default retention
period.
A default retention period will be configured for ${bucketPrefix}-retention-
after-creation.` ,
        { preformatted: true },
    );

/**
 * @param {Scenarios} scenarios
 */
const confirmUpdateRetention = (scenarios) =>
    new scenarios.ScenarioInput(
        "confirmUpdateRetention",
        "Configure default retention period?",
        { type: "confirm" },
    );
```

```
/**
 * @param {Scenarios} scenarios
 * @param {S3Client} client
 */
const updateRetentionAction = (scenarios, client) =>
  new scenarios.ScenarioAction("updateRetentionAction", async (state) => {
    await client.send(
      new PutBucketVersioningCommand({
        Bucket: state.retentionBucketName,
        VersioningConfiguration: {
          MFADelete: MFADeleteStatus.Disabled,
          Status: BucketVersioningStatus.Enabled,
        },
      }),
    );

    await client.send(
      new PutObjectLockConfigurationCommand({
        Bucket: state.retentionBucketName,
        ObjectLockConfiguration: {
          ObjectLockEnabled: "Enabled",
          Rule: {
            DefaultRetention: {
              Mode: "GOVERNANCE",
              Years: 1,
            },
          },
        },
      }),
    );
  });

/**
 * @param {Scenarios} scenarios
 */
const updateLockPolicy = (scenarios) =>
  new scenarios.ScenarioOutput(
    "updateLockPolicy",
    `Object lock policies can also be added to existing buckets.
    An object lock policy will be added to ${bucketPrefix}-lock-enabled.`,
    { preformatted: true },
  );
```

```
/**
 * @param {Scenarios} scenarios
 */
const confirmUpdateLockPolicy = (scenarios) =>
  new scenarios.ScenarioInput(
    "confirmUpdateLockPolicy",
    "Add object lock policy?",
    { type: "confirm" },
  );

/**
 * @param {Scenarios} scenarios
 * @param {S3Client} client
 */
const updateLockPolicyAction = (scenarios, client) =>
  new scenarios.ScenarioAction("updateLockPolicyAction", async (state) => {
    await client.send(
      new PutObjectLockConfigurationCommand({
        Bucket: state.lockEnabledBucketName,
        ObjectLockConfiguration: {
          ObjectLockEnabled: "Enabled",
        },
      }),
    );
  });

/**
 * @param {Scenarios} scenarios
 * @param {S3Client} client
 */
const confirmSetLegalHoldFileEnabled = (scenarios) =>
  new scenarios.ScenarioInput(
    "confirmSetLegalHoldFileEnabled",
    (state) =>
      `Would you like to add a legal hold to file0.txt in
      ${state.lockEnabledBucketName}?`,
    {
      type: "confirm",
    },
  );

/**
 * @param {Scenarios} scenarios
 * @param {S3Client} client
```

```

*/
const setLegalHoldFileEnabledAction = (scenarios, client) =>
  new scenarios.ScenarioAction(
    "setLegalHoldFileEnabledAction",
    async (state) => {
      await client.send(
        new PutObjectLegalHoldCommand({
          Bucket: state.lockEnabledBucketName,
          Key: "file0.txt",
          LegalHold: {
            Status: ObjectLockLegalHoldStatus.ON,
          },
        }),
      );
      console.log(
        `Modified legal hold for file0.txt in ${state.lockEnabledBucketName}.`,
      );
    },
    { skipWhen: (state) => !state.confirmSetLegalHoldFileEnabled },
  );

/**
 * @param {Scenarios} scenarios
 * @param {S3Client} client
 */
const confirmSetRetentionPeriodFileEnabled = (scenarios) =>
  new scenarios.ScenarioInput(
    "confirmSetRetentionPeriodFileEnabled",
    (state) =>
      `Would you like to add a 1 day Governance retention period to file1.txt in
      ${state.lockEnabledBucketName}?
      Reminder: Only a user with the s3:BypassGovernanceRetention permission will be able
      to delete this file or its bucket until the retention period has expired.`,
    {
      type: "confirm",
    },
  );

/**
 * @param {Scenarios} scenarios
 * @param {S3Client} client
 */
const setRetentionPeriodFileEnabledAction = (scenarios, client) =>
  new scenarios.ScenarioAction(

```

```
"setRetentionPeriodFileEnabledAction",
async (state) => {
  const retentionDate = new Date();
  retentionDate.setDate(retentionDate.getDate() + 1);
  await client.send(
    new PutObjectRetentionCommand({
      Bucket: state.lockEnabledBucketName,
      Key: "file1.txt",
      Retention: {
        Mode: ObjectLockRetentionMode.GOVERNANCE,
        RetainUntilDate: retentionDate,
      },
    }),
  );
  console.log(
    `Set retention for file1.txt in ${state.lockEnabledBucketName} until
    ${retentionDate.toISOString().split("T")[0]}.`,
  );
},
{ skipWhen: (state) => !state.confirmSetRetentionPeriodFileEnabled },
);

/**
 * @param {Scenarios} scenarios
 * @param {S3Client} client
 */
const confirmSetLegalHoldFileRetention = (scenarios) =>
  new scenarios.ScenarioInput(
    "confirmSetLegalHoldFileRetention",
    (state) =>
      `Would you like to add a legal hold to file0.txt in
      ${state.retentionBucketName}?`,
    {
      type: "confirm",
    },
  );

/**
 * @param {Scenarios} scenarios
 * @param {S3Client} client
 */
const setLegalHoldFileRetentionAction = (scenarios, client) =>
  new scenarios.ScenarioAction(
    "setLegalHoldFileRetentionAction",
```

```

    async (state) => {
      await client.send(
        new PutObjectLegalHoldCommand({
          Bucket: state.retentionBucketName,
          Key: "file0.txt",
          LegalHold: {
            Status: ObjectLockLegalHoldStatus.ON,
          },
        }),
      );
      console.log(
        `Modified legal hold for file0.txt in ${state.retentionBucketName}.`,
      );
    },
    { skipWhen: (state) => !state.confirmSetLegalHoldFileRetention },
  );

/**
 * @param {Scenarios} scenarios
 */
const confirmSetRetentionPeriodFileRetention = (scenarios) =>
  new scenarios.ScenarioInput(
    "confirmSetRetentionPeriodFileRetention",
    (state) =>
      `Would you like to add a 1 day Governance retention period to file1.txt in
      ${state.retentionBucketName}?
      Reminder: Only a user with the s3:BypassGovernanceRetention permission will be able
      to delete this file or its bucket until the retention period has expired.`,
    {
      type: "confirm",
    },
  );

/**
 * @param {Scenarios} scenarios
 * @param {S3Client} client
 */
const setRetentionPeriodFileRetentionAction = (scenarios, client) =>
  new scenarios.ScenarioAction(
    "setRetentionPeriodFileRetentionAction",
    async (state) => {
      const retentionDate = new Date();
      retentionDate.setDate(retentionDate.getDate() + 1);
      await client.send(

```

```
    new PutObjectRetentionCommand({
      Bucket: state.retentionBucketName,
      Key: "file1.txt",
      Retention: {
        Mode: ObjectLockRetentionMode.GOVERNANCE,
        RetainUntilDate: retentionDate,
      },
      BypassGovernanceRetention: true,
    }),
  );
  console.log(
    `Set retention for file1.txt in ${state.retentionBucketName} until
    ${retentionDate.toISOString().split("T")[0]}.`,
  );
},
{ skipWhen: (state) => !state.confirmSetRetentionPeriodFileRetention },
);

export {
  createBuckets,
  confirmCreateBuckets,
  createBucketsAction,
  populateBuckets,
  confirmPopulateBuckets,
  populateBucketsAction,
  updateRetention,
  confirmUpdateRetention,
  updateRetentionAction,
  updateLockPolicy,
  confirmUpdateLockPolicy,
  updateLockPolicyAction,
  confirmSetLegalHoldFileEnabled,
  setLegalHoldFileEnabledAction,
  confirmSetRetentionPeriodFileEnabled,
  setRetentionPeriodFileEnabledAction,
  confirmSetLegalHoldFileRetention,
  setLegalHoldFileRetentionAction,
  confirmSetRetentionPeriodFileRetention,
  setRetentionPeriodFileRetentionAction,
};
```

[repl.steps.js](#) - View and delete files in the buckets.


```
import {
  ChecksumAlgorithm,
  DeleteObjectCommand,
  GetObjectLegalHoldCommand,
  GetObjectLockConfigurationCommand,
  GetObjectRetentionCommand,
  ListObjectVersionsCommand,
  PutObjectCommand,
} from "@aws-sdk/client-s3";

/**
 * @typedef {import("@aws-doc-sdk-examples/lib/scenario/index.js")} Scenarios
 */

/**
 * @typedef {import("@aws-sdk/client-s3").S3Client} S3Client
 */

const choices = {
  EXIT: 0,
  LIST_ALL_FILES: 1,
  DELETE_FILE: 2,
  DELETE_FILE_WITH_RETENTION: 3,
  OVERWRITE_FILE: 4,
  VIEW_RETENTION_SETTINGS: 5,
  VIEW_LEGAL_HOLD_SETTINGS: 6,
};

/**
 * @param {Scenarios} scenarios
 */
const replInput = (scenarios) =>
  new scenarios.ScenarioInput(
    "replChoice",
    `Explore the S3 locking features by selecting one of the following choices`,
    {
      type: "select",
      choices: [
        { name: "List all files in buckets", value: choices.LIST_ALL_FILES },
        { name: "Attempt to delete a file.", value: choices.DELETE_FILE },
        {
          name: "Attempt to delete a file with retention period bypass.",
          value: choices.DELETE_FILE_WITH_RETENTION,
        }
      ]
    }
  )
```

```

    },
    { name: "Attempt to overwrite a file.", value: choices.OVERWRITE_FILE },
    {
      name: "View the object and bucket retention settings for a file.",
      value: choices.VIEW_RETENTION_SETTINGS,
    },
    {
      name: "View the legal hold settings for a file.",
      value: choices.VIEW_LEGAL_HOLD_SETTINGS,
    },
    { name: "Finish the workflow.", value: choices.EXIT },
  ],
},
);

/**
 * @param {S3Client} client
 * @param {string[]} buckets
 */
const getAllFiles = async (client, buckets) => {
  /** @type {{bucket: string, key: string, version: string}[]} */
  const files = [];
  for (const bucket of buckets) {
    const objectsResponse = await client.send(
      new ListObjectVersionsCommand({ Bucket: bucket }),
    );
    for (const version of objectsResponse.Versions || []) {
      const { Key, VersionId } = version;
      files.push({ bucket, key: Key, version: VersionId });
    }
  }

  return files;
};

/**
 * @param {Scenarios} scenarios
 * @param {S3Client} client
 */
const replAction = (scenarios, client) =>
  new scenarios.ScenarioAction(
    "replAction",
    async (state) => {
      const files = await getAllFiles(client, [

```

```
    state.noLockBucketName,
    state.lockEnabledBucketName,
    state.retentionBucketName,
  ]);

const fileInput = new scenarios.ScenarioInput(
  "selectedFile",
  "Select a file:",
  {
    type: "select",
    choices: files.map((file, index) => ({
      name: `${index + 1}: ${file.bucket}: ${file.key} (version: ${
        file.version
      })`,
      value: index,
    })),
  },
);

const { replChoice } = state;

switch (replChoice) {
  case choices.LIST_ALL_FILES: {
    const files = await getAllFiles(client, [
      state.noLockBucketName,
      state.lockEnabledBucketName,
      state.retentionBucketName,
    ]);
    state.replOutput = files
      .map(
        (file) =>
          `${file.bucket}: ${file.key} (version: ${file.version})`,
      )
      .join("\n");
    break;
  }
  case choices.DELETE_FILE: {
    /** @type {number} */
    const fileToDelete = await fileInput.handle(state);
    const selectedFile = files[fileToDelete];
    try {
      await client.send(
        new DeleteObjectCommand({
          Bucket: selectedFile.bucket,
```

```
        Key: selectedFile.key,
        VersionId: selectedFile.version,
    )),
    );
    state.replOutput = `Deleted ${selectedFile.key} in
${selectedFile.bucket}.`;
    } catch (err) {
        state.replOutput = `Unable to delete object ${selectedFile.key} in
bucket ${selectedFile.bucket}: ${err.message}`;
    }
    break;
}
case choices.DELETE_FILE_WITH_RETENTION: {
    /** @type {number} */
    const fileToDelete = await fileInput.handle(state);
    const selectedFile = files[fileToDelete];
    try {
        await client.send(
            new DeleteObjectCommand({
                Bucket: selectedFile.bucket,
                Key: selectedFile.key,
                VersionId: selectedFile.version,
                BypassGovernanceRetention: true,
            )),
        );
        state.replOutput = `Deleted ${selectedFile.key} in
${selectedFile.bucket}.`;
    } catch (err) {
        state.replOutput = `Unable to delete object ${selectedFile.key} in
bucket ${selectedFile.bucket}: ${err.message}`;
    }
    break;
}
case choices.OVERWRITE_FILE: {
    /** @type {number} */
    const fileToOverwrite = await fileInput.handle(state);
    const selectedFile = files[fileToOverwrite];
    try {
        await client.send(
            new PutObjectCommand({
                Bucket: selectedFile.bucket,
                Key: selectedFile.key,
                Body: "New content",
                ChecksumAlgorithm: ChecksumAlgorithm.SHA256,
```

```

        })),
    );
    state.replOutput = `Overwrote ${selectedFile.key} in
${selectedFile.bucket}.`;
    } catch (err) {
        state.replOutput = `Unable to overwrite object ${selectedFile.key} in
bucket ${selectedFile.bucket}: ${err.message}`;
    }
    break;
}
case choices.VIEW_RETENTION_SETTINGS: {
    /** @type {number} */
    const fileToView = await fileInput.handle(state);
    const selectedFile = files[fileToView];
    try {
        const retention = await client.send(
            new GetObjectRetentionCommand({
                Bucket: selectedFile.bucket,
                Key: selectedFile.key,
                VersionId: selectedFile.version,
            })),
        );
        const bucketConfig = await client.send(
            new GetObjectLockConfigurationCommand({
                Bucket: selectedFile.bucket,
            })),
        );
        state.replOutput = `Object retention for ${selectedFile.key}
in ${selectedFile.bucket}: ${retention.Retention?.Mode} until
${retention.Retention?.RetainUntilDate?.toISOString()}.
Bucket object lock config for ${selectedFile.bucket} in ${selectedFile.bucket}:
Enabled: ${bucketConfig.ObjectLockConfiguration?.ObjectLockEnabled}
Rule:
${JSON.stringify(bucketConfig.ObjectLockConfiguration?.Rule?.DefaultRetention)}`;
    } catch (err) {
        state.replOutput = `Unable to fetch object lock retention:
'${err.message}'`;
    }
    break;
}
case choices.VIEW_LEGAL_HOLD_SETTINGS: {
    /** @type {number} */
    const fileToView = await fileInput.handle(state);
    const selectedFile = files[fileToView];

```

```

    try {
      const legalHold = await client.send(
        new GetObjectLegalHoldCommand({
          Bucket: selectedFile.bucket,
          Key: selectedFile.key,
          VersionId: selectedFile.version,
        }),
      );
      state.replOutput = `Object legal hold for ${selectedFile.key} in
${selectedFile.bucket}: Status: ${legalHold.LegalHold?.Status}`;
    } catch (err) {
      state.replOutput = `Unable to fetch legal hold: '${err.message}'`;
    }
    break;
  }
  default:
    throw new Error(`Invalid replChoice: ${replChoice}`);
  }
},
{
  whileConfig: {
    whileFn: ({ replChoice }) => replChoice !== choices.EXIT,
    input: replInput(scenarios),
    output: new scenarios.ScenarioOutput(
      "REPL output",
      (state) => state.replOutput,
      { preformatted: true },
    ),
  },
},
);

export { replInput, replAction, choices };

```

`clean.steps.js` - Destroy all created resources.

```

import {
  DeleteObjectCommand,
  DeleteBucketCommand,
  ListObjectVersionsCommand,
  GetObjectLegalHoldCommand,
  GetObjectRetentionCommand,

```

```
    PutObjectLegalHoldCommand,
  } from "@aws-sdk/client-s3";

/**
 * @typedef {import("@aws-doc-sdk-examples/lib/scenario/index.js")} Scenarios
 */

/**
 * @typedef {import("@aws-sdk/client-s3").S3Client} S3Client
 */

/**
 * @param {Scenarios} scenarios
 */
const confirmCleanup = (scenarios) =>
  new scenarios.ScenarioInput("confirmCleanup", "Clean up resources?", {
    type: "confirm",
  });

/**
 * @param {Scenarios} scenarios
 * @param {S3Client} client
 */
const cleanupAction = (scenarios, client) =>
  new scenarios.ScenarioAction("cleanupAction", async (state) => {
    const { noLockBucketName, lockEnabledBucketName, retentionBucketName } =
      state;

    const buckets = [
      noLockBucketName,
      lockEnabledBucketName,
      retentionBucketName,
    ];

    for (const bucket of buckets) {
      /** @type {import("@aws-sdk/client-s3").ListObjectVersionsCommandOutput} */
      let objectsResponse;

      try {
        objectsResponse = await client.send(
          new ListObjectVersionsCommand({
            Bucket: bucket,
          }),
        );
      }
    }
  });
```

```
    } catch (e) {
      if (e instanceof Error && e.name === "NoSuchBucket") {
        console.log("Object's bucket has already been deleted.");
        continue;
      } else {
        throw e;
      }
    }
  }

  for (const version of objectsResponse.Versions || []) {
    const { Key, VersionId } = version;

    try {
      const legalHold = await client.send(
        new GetObjectLegalHoldCommand({
          Bucket: bucket,
          Key,
          VersionId,
        })),
      );

      if (legalHold.LegalHold?.Status === "ON") {
        await client.send(
          new PutObjectLegalHoldCommand({
            Bucket: bucket,
            Key,
            VersionId,
            LegalHold: {
              Status: "OFF",
            },
          })),
        );
      }
    } catch (err) {
      console.log(
        `Unable to fetch legal hold for ${Key} in ${bucket}: '${err.message}'`,
      );
    }

    try {
      const retention = await client.send(
        new GetObjectRetentionCommand({
          Bucket: bucket,
          Key,
```



```
        VersionId,
      )),
    );

    if (retention.Retention?.Mode === "GOVERNANCE") {
      await client.send(
        new DeleteObjectCommand({
          Bucket: bucket,
          Key,
          VersionId,
          BypassGovernanceRetention: true,
        })),
      );
    }
  } catch (err) {
    console.log(
      `Unable to fetch object lock retention for ${Key} in ${bucket}:
    '${err.message}'`,
    );
  }

  await client.send(
    new DeleteObjectCommand({
      Bucket: bucket,
      Key,
      VersionId,
    })),
  );
}

await client.send(new DeleteBucketCommand({ Bucket: bucket }));
console.log(`Delete for ${bucket} complete.`);
}
});

export { confirmCleanup, cleanupAction };
```

- For API details, see the following topics in *AWS SDK for JavaScript API Reference*.
 - [GetObjectLegalHold](#)
 - [GetObjectLockConfiguration](#)
 - [GetObjectRetention](#)

- [PutObjectLegalHold](#)
- [PutObjectLockConfiguration](#)
- [PutObjectRetention](#)

Upload or download large files

The following code example shows how to upload or download large files to and from Amazon S3.

For more information, see [Uploading an object using multipart upload](#).

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Upload a large file.

```
import {
  CreateMultipartUploadCommand,
  UploadPartCommand,
  CompleteMultipartUploadCommand,
  AbortMultipartUploadCommand,
  S3Client,
} from "@aws-sdk/client-s3";

const twentyFiveMB = 25 * 1024 * 1024;

export const createString = (size = twentyFiveMB) => {
  return "x".repeat(size);
};

export const main = async () => {
  const s3Client = new S3Client({});
  const bucketName = "test-bucket";
  const key = "multipart.txt";
  const str = createString();
  const buffer = Buffer.from(str, "utf8");
```

```
let uploadId;

try {
  const multipartUpload = await s3Client.send(
    new CreateMultipartUploadCommand({
      Bucket: bucketName,
      Key: key,
    }),
  );

  uploadId = multipartUpload.UploadId;

  const uploadPromises = [];
  // Multipart uploads require a minimum size of 5 MB per part.
  const partSize = Math.ceil(buffer.length / 5);

  // Upload each part.
  for (let i = 0; i < 5; i++) {
    const start = i * partSize;
    const end = start + partSize;
    uploadPromises.push(
      s3Client
        .send(
          new UploadPartCommand({
            Bucket: bucketName,
            Key: key,
            UploadId: uploadId,
            Body: buffer.subarray(start, end),
            PartNumber: i + 1,
          }),
        )
        .then((d) => {
          console.log("Part", i + 1, "uploaded");
          return d;
        }),
    );
  }

  const uploadResults = await Promise.all(uploadPromises);

  return await s3Client.send(
    new CompleteMultipartUploadCommand({
      Bucket: bucketName,
      Key: key,
```

```

    UploadId: uploadId,
    MultipartUpload: {
      Parts: uploadResults.map(({ ETag }, i) => ({
        ETag,
        PartNumber: i + 1,
      })),
    },
  })),
);

// Verify the output by downloading the file from the Amazon Simple Storage
Service (Amazon S3) console.
// Because the output is a 25 MB string, text editors might struggle to open the
file.
} catch (err) {
  console.error(err);

  if (uploadId) {
    const abortCommand = new AbortMultipartUploadCommand({
      Bucket: bucketName,
      Key: key,
      UploadId: uploadId,
    });

    await s3Client.send(abortCommand);
  }
}
};

```

Download a large file.

```

import { GetObjectCommand, S3Client } from "@aws-sdk/client-s3";
import { createWriteStream } from "fs";

const s3Client = new S3Client({});
const oneMB = 1024 * 1024;

export const getObjectRange = ({ bucket, key, start, end }) => {
  const command = new GetObjectCommand({
    Bucket: bucket,
    Key: key,
    Range: `bytes=${start}-${end}`,
  });

```

```
});

return s3Client.send(command);
};

/**
 * @param {string | undefined} contentRange
 */
export const getRangeAndLength = (contentRange) => {
  const [range, length] = contentRange.split("/");
  const [start, end] = range.split("-");
  return {
    start: parseInt(start),
    end: parseInt(end),
    length: parseInt(length),
  };
};

export const isComplete = ({ end, length }) => end === length - 1;

// When downloading a large file, you might want to break it down into
// smaller pieces. Amazon S3 accepts a Range header to specify the start
// and end of the byte range to be downloaded.
const downloadInChunks = async ({ bucket, key }) => {
  const writeStream = createWriteStream(
    fileURLToPath(new URL(`./${key}`, import.meta.url)),
  ).on("error", (err) => console.error(err));

  let rangeAndLength = { start: -1, end: -1, length: -1 };

  while (!isComplete(rangeAndLength)) {
    const { end } = rangeAndLength;
    const nextRange = { start: end + 1, end: end + oneMB };

    console.log(`Downloading bytes ${nextRange.start} to ${nextRange.end}`);

    const { ContentRange, Body } = await getObjectRange({
      bucket,
      key,
      ...nextRange,
    });

    writeStream.write(await Body.transformToByteArray());
    rangeAndLength = getRangeAndLength(ContentRange);
  }
};
```

```
    }  
  };  
  
  export const main = async () => {  
    await downloadInChunks({  
      bucket: "my-cool-bucket",  
      key: "my-cool-object.txt",  
    });  
  };  
};
```

Serverless examples

Invoke a Lambda function from an Amazon S3 trigger

The following code example shows how to implement a Lambda function that receives an event triggered by uploading an object to an S3 bucket. The function retrieves the S3 bucket name and object key from the event parameter and calls the Amazon S3 API to retrieve and log the content type of the object.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [Serverless examples](#) repository.

Consuming an S3 event with Lambda using JavaScript.

```
import { S3Client, HeadObjectCommand } from "@aws-sdk/client-s3";  
  
const client = new S3Client();  
  
export const handler = async (event, context) => {  
  
  // Get the object from the event and show its content type  
  const bucket = event.Records[0].s3.bucket.name;  
  const key = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/g, '  
'));
```

```
    try {
      const { ContentType } = await client.send(new HeadObjectCommand({
        Bucket: bucket,
        Key: key,
      }));

      console.log('CONTENT TYPE:', ContentType);
      return ContentType;

    } catch (err) {
      console.log(err);
      const message = `Error getting object ${key} from bucket ${bucket}. Make
sure they exist and your bucket is in the same region as this function.`;
      console.log(message);
      throw new Error(message);
    }
  };
};
```

Consuming an S3 event with Lambda using TypeScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { S3Event } from 'aws-lambda';
import { S3Client, HeadObjectCommand } from '@aws-sdk/client-s3';

const s3 = new S3Client({ region: process.env.AWS_REGION });

export const handler = async (event: S3Event): Promise<string | undefined> => {
  // Get the object from the event and show its content type
  const bucket = event.Records[0].s3.bucket.name;
  const key = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/g, '
'));
  const params = {
    Bucket: bucket,
    Key: key,
  };
  try {
    const { ContentType } = await s3.send(new HeadObjectCommand(params));
    console.log('CONTENT TYPE:', ContentType);
    return ContentType;
  } catch (err) {
    console.log(err);
  }
};
```

```
const message = `Error getting object ${key} from bucket ${bucket}. Make sure
they exist and your bucket is in the same region as this function.`;
console.log(message);
throw new Error(message);
}
};
```

S3 Glacier examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with S3 Glacier.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Topics

- [Actions](#)

Actions

CreateVault

The following code example shows how to use `CreateVault`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create the client.

```
const { GlacierClient } = require("@aws-sdk/client-glacier");
// Set the AWS Region.
```



```
const REGION = "REGION";
//Set the Redshift Service Object
const glacierClient = new GlacierClient({ region: REGION });
export { glacierClient };
```

Create the vault.

```
// Load the SDK for JavaScript
import { CreateVaultCommand } from "@aws-sdk/client-glacier";
import { glacierClient } from "../libs/glacierClient.js";

// Set the parameters
const vaultname = "VAULT_NAME"; // VAULT_NAME
const params = { vaultName: vaultname };

const run = async () => {
  try {
    const data = await glacierClient.send(new CreateVaultCommand(params));
    console.log("Success, vault created!");
    return data; // For unit tests.
  } catch (err) {
    console.log("Error");
  }
};
run();
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [CreateVault](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// Load the SDK for JavaScript
var AWS = require("aws-sdk");
// Set the region
```

```
AWS.config.update({ region: "REGION" });

// Create a new service object
var glacier = new AWS.Glacier({ apiVersion: "2012-06-01" });
// Call Glacier to create the vault
glacier.createVault({ vaultName: "YOUR_VAULT_NAME" }, function (err) {
  if (!err) {
    console.log("Created vault!");
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [CreateVault](#) in *AWS SDK for JavaScript API Reference*.

UploadArchive

The following code example shows how to use UploadArchive.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create the client.

```
const { GlacierClient } = require("@aws-sdk/client-glacier");
// Set the AWS Region.
const REGION = "REGION";
//Set the Redshift Service Object
const glacierClient = new GlacierClient({ region: REGION });
export { glacierClient };
```

Upload the archive.

```
// Load the SDK for JavaScript
```

```
import { UploadArchiveCommand } from "@aws-sdk/client-glacier";
import { glacierClient } from "../libs/glacierClient.js";

// Set the parameters
const vaultname = "VAULT_NAME"; // VAULT_NAME

// Create a new service object and buffer
const buffer = new Buffer.alloc(2.5 * 1024 * 1024); // 2.5MB buffer
const params = { vaultName: vaultname, body: buffer };

const run = async () => {
  try {
    const data = await glacierClient.send(new UploadArchiveCommand(params));
    console.log("Archive ID", data.archiveId);
    return data; // For unit tests.
  } catch (err) {
    console.log("Error uploading archive!", err);
  }
};
run();
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [UploadArchive](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

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

// Create a new service object and buffer
var glacier = new AWS.Glacier({ apiVersion: "2012-06-01" });
buffer = Buffer.alloc(2.5 * 1024 * 1024); // 2.5MB buffer
```

```
var params = { vaultName: "YOUR_VAULT_NAME", body: buffer };
// Call Glacier to upload the archive.
glacier.uploadArchive(params, function (err, data) {
  if (err) {
    console.log("Error uploading archive!", err);
  } else {
    console.log("Archive ID", data.archiveId);
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [UploadArchive](#) in *AWS SDK for JavaScript API Reference*.

SageMaker examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with SageMaker.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Scenarios are code examples that show you how to accomplish specific tasks by calling multiple functions within a service or combined with other AWS services.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Get started

Hello SageMaker

The following code examples show how to get started using SageMaker.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import {
  SageMakerClient,
  ListNotebookInstancesCommand,
} from "@aws-sdk/client-sagemaker";

const client = new SageMakerClient({
  region: "us-west-2",
});

export const helloSagemaker = async () => {
  const command = new ListNotebookInstancesCommand({ MaxResults: 5 });

  const response = await client.send(command);
  console.log(
    "Hello Amazon SageMaker! Let's list some of your notebook instances:",
  );

  const instances = response.NotebookInstances || [];

  if (instances.length === 0) {
    console.log(
      "• No notebook instances found. Try creating one in the AWS Management Console  
or with the CreateNotebookInstanceCommand.",
    );
  } else {
    console.log(
      instances
        .map(
          (i) =>
            `• Instance: ${i.NotebookInstanceName}\n  Arn:${i.NotebookInstanceArn} \n  Creation Date: ${i.CreationTime.toISOString()}`,
        )
        .join("\n"),
    );
  }

  return response;
};
```

- For API details, see [ListNotebookInstances](#) in *AWS SDK for JavaScript API Reference*.

Topics

- [Actions](#)
- [Scenarios](#)

Actions

CreatePipeline

The following code example shows how to use CreatePipeline.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

A function that creates a SageMaker pipeline using a locally provided JSON definition.

```
/**
 * Create the Amazon SageMaker pipeline using a JSON pipeline definition. The
 * definition
 * can also be provided as an Amazon S3 object using PipelineDefinitionS3Location.
 * @param {{roleArn: string, name: string, sagemakerClient: import('@aws-sdk/client-
 * sagemaker').SageMakerClient}} props
 */
export async function createSagemakerPipeline({
  // Assumes an AWS IAM role has been created for this pipeline.
  roleArn,
  name,
  // Assumes an AWS Lambda function has been created for this pipeline.
  functionArn,
  sagemakerClient,
}) {
  const pipelineDefinition = readFileSync(
    // dirnameFromMetaUrl is a local utility function. You can find its
    implementation
    // on GitHub.
    `${dirnameFromMetaUrl(
      import.meta.url,
```

```
    })..../workflows/sagemaker_pipelines/resources/
GeoSpatialPipeline.json`,
  )
  .toString()
  .replace(/\*FUNCTION_ARN\*/g, functionArn);

let arn = null;

const createPipeline = () =>
  sagemakerClient.send(
    new CreatePipelineCommand({
      PipelineName: name,
      PipelineDefinition: pipelineDefinition,
      RoleArn: roleArn,
    }),
  );

try {
  const { PipelineArn } = await createPipeline();
  arn = PipelineArn;
} catch (caught) {
  if (
    caught instanceof Error &&
    caught.name === "ValidationException" &&
    caught.message.includes(
      "Pipeline names must be unique within an AWS account and region",
    )
  ) {
    const { PipelineArn } = await sagemakerClient.send(
      new DescribePipelineCommand({ PipelineName: name }),
    );
    arn = PipelineArn;
  } else {
    throw caught;
  }
}

return {
  arn,
  cleanUp: async () => {
    await sagemakerClient.send(
      new DeletePipelineCommand({ PipelineName: name }),
    );
  },
}
```

```
};  
}
```

- For API details, see [CreatePipeline](#) in *AWS SDK for JavaScript API Reference*.

DeletePipeline

The following code example shows how to use DeletePipeline.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

The syntax for deleting a SageMaker pipeline. This code is part of a larger function. Refer to 'Create a pipeline' or the GitHub repository for more context.

```
await sagemakerClient.send(  
  new DeletePipelineCommand({ PipelineName: name } ),  
);
```

- For API details, see [DeletePipeline](#) in *AWS SDK for JavaScript API Reference*.

DescribePipelineExecution

The following code example shows how to use DescribePipelineExecution.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Wait for a SageMaker pipeline execution to succeed, fail, or stop.

```
/**
 * Poll the executing pipeline until the status is 'SUCCEEDED', 'STOPPED', or
 * 'FAILED'.
 * @param {{ arn: string, sagemakerClient: import('@aws-sdk/client-
 * sagemaker').SageMakerClient, wait: (ms: number) => Promise<void>}} props
 */
export async function waitForPipelineComplete({ arn, sagemakerClient, wait }) {
  const command = new DescribePipelineExecutionCommand({
    PipelineExecutionArn: arn,
  });

  let complete = false;
  let intervalInSeconds = 15;
  const COMPLETION_STATUSES = [
    PipelineExecutionStatus.FAILED,
    PipelineExecutionStatus.STOPPED,
    PipelineExecutionStatus.SUCCEEDED,
  ];

  do {
    const { PipelineExecutionStatus: status, FailureReason } =
      await sagemakerClient.send(command);

    complete = COMPLETION_STATUSES.includes(status);

    if (!complete) {
      console.log(
        `Pipeline is ${status}. Waiting ${intervalInSeconds} seconds before checking
        again.`
      );
      await wait(intervalInSeconds);
    } else if (status === PipelineExecutionStatus.FAILED) {
      throw new Error(`Pipeline failed because: ${FailureReason}`);
    } else if (status === PipelineExecutionStatus.STOPPED) {
      throw new Error(`Pipeline was forcefully stopped.`);
    } else {
      console.log(`Pipeline execution ${status}.`);
    }
  } while (!complete);
}
```

- For API details, see [DescribePipelineExecution](#) in *AWS SDK for JavaScript API Reference*.

StartPipelineExecution

The following code example shows how to use StartPipelineExecution.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Start a SageMaker pipeline execution.

```
/**
 * Start the execution of the Amazon SageMaker pipeline. Parameters that are
 * passed in are used in the AWS Lambda function.
 * @param {{
 *   name: string,
 *   sagemakerClient: import('@aws-sdk/client-sagemaker').SageMakerClient,
 *   roleArn: string,
 *   queueUrl: string,
 *   s3InputBucketName: string,
 * }} props
 */
export async function startPipelineExecution({
  sagemakerClient,
  name,
  bucketName,
  roleArn,
  queueUrl,
}) {
  /**
   * The Vector Enrichment Job requests CSV data. This configuration points to a CSV
   * file in an Amazon S3 bucket.
   * @type {import("@aws-sdk/client-sagemaker-geospatial").VectorEnrichmentJobInputConfig}
   */
  const inputConfig = {
    DataSourceConfig: {
```

```
    S3Data: {
      S3Uri: `s3://${bucketName}/input/sample_data.csv`,
    },
  },
  DocumentType: VectorEnrichmentJobDocumentType.CSV,
};

/**
 * The Vector Enrichment Job adds additional data to the source CSV. This
configuration points
 * to an Amazon S3 prefix where the output will be stored.
 * @type {import("@aws-sdk/client-sagemaker-geospatial").ExportVectorEnrichmentJobOutputConfig}
 */
const outputConfig = {
  S3Data: {
    S3Uri: `s3://${bucketName}/output/`,
  },
};

/**
 * This job will be a Reverse Geocoding Vector Enrichment Job. Reverse Geocoding
requires
 * latitude and longitude values.
 * @type {import("@aws-sdk/client-sagemaker-geospatial").VectorEnrichmentJobConfig}
 */
const jobConfig = {
  ReverseGeocodingConfig: {
    XAttributeName: "Longitude",
    YAttributeName: "Latitude",
  },
};

const { PipelineExecutionArn } = await sagemakerClient.send(
  new StartPipelineExecutionCommand({
    PipelineName: name,
    PipelineExecutionDisplayName: `${name}-example-execution`,
    PipelineParameters: [
      { Name: "parameter_execution_role", Value: roleArn },
      { Name: "parameter_queue_url", Value: queueUrl },
      {
        Name: "parameter_vej_input_config",
        Value: JSON.stringify(inputConfig),
      }
    ]
  })
);
```

```
    },
    {
      Name: "parameter_vej_export_config",
      Value: JSON.stringify(outputConfig),
    },
    {
      Name: "parameter_step_1_vej_config",
      Value: JSON.stringify(jobConfig),
    },
  ],
 )),
);

return {
  arn: PipelineExecutionArn,
};
}
```

- For API details, see [StartPipelineExecution](#) in *AWS SDK for JavaScript API Reference*.

Scenarios

Get started with geospatial jobs and pipelines

The following code example shows how to:

- Set up resources for a pipeline.
- Set up a pipeline that executes a geospatial job.
- Start a pipeline execution.
- Monitor the status of the execution.
- View the output of the pipeline.
- Clean up resources.

For more information, see [Create and run SageMaker pipelines using AWS SDKs on Community.aws](#).

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

The following file excerpt contains functions that use the SageMaker client to manage a pipeline.

```
import { readFileSync } from "fs";

import {
  CreateRoleCommand,
  DeleteRoleCommand,
  CreatePolicyCommand,
  DeletePolicyCommand,
  AttachRolePolicyCommand,
  DetachRolePolicyCommand,
  GetRoleCommand,
  ListPoliciesCommand,
} from "@aws-sdk/client-iam";

import {
  PublishLayerVersionCommand,
  DeleteLayerVersionCommand,
  CreateFunctionCommand,
  Runtime,
  DeleteFunctionCommand,
  CreateEventSourceMappingCommand,
  DeleteEventSourceMappingCommand,
  GetFunctionCommand,
} from "@aws-sdk/client-lambda";

import {
  PutObjectCommand,
  CreateBucketCommand,
  DeleteBucketCommand,
  DeleteObjectCommand,
  GetObjectCommand,
  ListObjectsV2Command,
```

```
} from "@aws-sdk/client-s3";

import {
  CreatePipelineCommand,
  DeletePipelineCommand,
  DescribePipelineCommand,
  DescribePipelineExecutionCommand,
  PipelineExecutionStatus,
  StartPipelineExecutionCommand,
} from "@aws-sdk/client-sagemaker";

import { VectorEnrichmentJobDocumentType } from "@aws-sdk/client-sagemaker-geospatial";

import {
  CreateQueueCommand,
  DeleteQueueCommand,
  GetQueueAttributesCommand,
  GetQueueUrlCommand,
} from "@aws-sdk/client-sqs";

import { dirnameFromMetaUrl } from "@aws-doc-sdk-examples/lib/utils/util-fs.js";
import { retry } from "@aws-doc-sdk-examples/lib/utils/util-timers.js";

/**
 * Create the AWS IAM role that will be assumed by AWS Lambda.
 * @param {{ name: string, iamClient: import('@aws-sdk/client-iam').IAMClient }}
 * props
 */
export async function createLambdaExecutionRole({ name, iamClient }) {
  const createRole = () =>
    iamClient.send(
      new CreateRoleCommand({
        RoleName: name,
        AssumeRolePolicyDocument: JSON.stringify({
          Version: "2012-10-17",
          Statement: [
            {
              Effect: "Allow",
              Action: ["sts:AssumeRole"],
              Principal: { Service: ["lambda.amazonaws.com"] },
            },
          ],
        }),
      }),
    ),
}
```

```

    }),
  );

  let role = null;

  try {
    const { Role } = await createRole();
    role = Role;
  } catch (caught) {
    if (
      caught instanceof Error &&
      caught.name === "EntityAlreadyExistsException"
    ) {
      const { Role } = await iamClient.send(
        new GetRoleCommand({ RoleName: name }),
      );
      role = Role;
    } else {
      throw caught;
    }
  }

  return {
    arn: role.Arn,
    cleanUp: async () => {
      await iamClient.send(new DeleteRoleCommand({ RoleName: name }));
    },
  };
}

/**
 * Create an AWS IAM policy that will be attached to the AWS IAM role assumed by the
 * AWS Lambda function.
 * The policy grants permission to work with Amazon SQS, Amazon CloudWatch, and
 * Amazon SageMaker.
 * @param {{name: string, iamClient: import('@aws-sdk/client-iam').IAMClient,
 * pipelineExecutionRoleArn: string}} props
 */
export async function createLambdaExecutionPolicy({
  name,
  iamClient,
  pipelineExecutionRoleArn,
}) {
  const policyConfig = {

```

```

Version: "2012-10-17",
Statement: [
  {
    Effect: "Allow",
    Action: [
      "sqs:ReceiveMessage",
      "sqs>DeleteMessage",
      "sqs:GetQueueAttributes",
      "logs:CreateLogGroup",
      "logs:CreateLogStream",
      "logs:PutLogEvents",
      "sagemaker-geospatial:StartVectorEnrichmentJob",
      "sagemaker-geospatial:GetVectorEnrichmentJob",
      "sagemaker:SendPipelineExecutionStepFailure",
      "sagemaker:SendPipelineExecutionStepSuccess",
      "sagemaker-geospatial:ExportVectorEnrichmentJob",
    ],
    Resource: "*",
  },
  {
    Effect: "Allow",
    // The AWS Lambda function needs permission to pass the pipeline execution
    role to
    // the StartVectorEnrichmentCommand. This restriction prevents an AWS Lambda
    function
    // from elevating privileges. For more information, see:
    // https://docs.aws.amazon.com/IAM/latest/UserGuide/
    id_roles_use_passrole.html
    Action: ["iam:PassRole"],
    Resource: `${pipelineExecutionRoleArn}`,
    Condition: {
      StringEquals: {
        "iam:PassedToService": [
          "sagemaker.amazonaws.com",
          "sagemaker-geospatial.amazonaws.com",
        ],
      },
    },
  },
],
];

const createPolicy = () =>
  iamClient.send(

```



```
    new CreatePolicyCommand({
      PolicyDocument: JSON.stringify(policyConfig),
      PolicyName: name,
    }),
  );

let policy = null;

try {
  const { Policy } = await createPolicy();
  policy = Policy;
} catch (caught) {
  if (
    caught instanceof Error &&
    caught.name === "EntityAlreadyExistsException"
  ) {
    const { Policies } = await iamClient.send(new ListPoliciesCommand({}));
    if (Policies) {
      policy = Policies.find((p) => p.PolicyName === name);
    } else {
      throw new Error("No policies found.");
    }
  } else {
    throw caught;
  }
}

return {
  arn: policy?.Arn,
  policyConfig,
  cleanUp: async () => {
    await iamClient.send(new DeletePolicyCommand({ PolicyArn: policy?.Arn }));
  },
};
}

/**
 * Attach an AWS IAM policy to an AWS IAM role.
 * @param {{roleName: string, policyArn: string, iamClient: import('@aws-sdk/client-iam').IAMClient}} props
 */
export async function attachPolicy({ roleName, policyArn, iamClient }) {
  const attachPolicyCommand = new AttachRolePolicyCommand({
    RoleName: roleName,
```

```

    PolicyArn: policyArn,
  });

  await iamClient.send(attachPolicyCommand);
  return {
    cleanUp: async () => {
      await iamClient.send(
        new DetachRolePolicyCommand({
          RoleName: roleName,
          PolicyArn: policyArn,
        })),
    };
  },
};
}

/**
 * Create an AWS Lambda layer that contains the Amazon SageMaker and Amazon
 * SageMaker Geospatial clients
 * in the runtime. The default runtime supports v3.188.0 of the JavaScript SDK. The
 * Amazon SageMaker
 * Geospatial client wasn't introduced until v3.221.0.
 * @param {{ name: string, lambdaClient: import('@aws-sdk/client-
 * lambda').LambdaClient }} props
 */
export async function createLambdaLayer({ name, lambdaClient }) {
  const layerPath = `${dirnameFromMetaUrl(import.meta.url)}lambda/nodejs.zip`;
  const { LayerVersionArn, Version } = await lambdaClient.send(
    new PublishLayerVersionCommand({
      LayerName: name,
      Content: {
        ZipFile: Uint8Array.from(readFileSync(layerPath)),
      },
    })),
  );

  return {
    versionArn: LayerVersionArn,
    version: Version,
    cleanUp: async () => {
      await lambdaClient.send(
        new DeleteLayerVersionCommand({
          LayerName: name,
          VersionNumber: Version,
        })),
    };
  };
}

```

```

    }},
  );
},
};
}

/**
 * Deploy the AWS Lambda function that will be used to respond to Amazon SageMaker
 * pipeline
 * execution steps.
 * @param {{roleArn: string, name: string, lambdaClient: import('@aws-sdk/client-
 * lambda').LambdaClient, layerVersionArn: string}} props
 */
export async function createLambdaFunction({
  name,
  roleArn,
  lambdaClient,
  layerVersionArn,
}) {
  const lambdaPath = `${dirnameFromMetaUrl(
    import.meta.url,
  )}lambda/dist/index.mjs.zip`;

  // If a function of the same name already exists, return that
  // function's ARN instead. By default this is
  // "sagemaker-wkflw-lambda-function", so collisions are
  // unlikely.
  const createFunction = async () => {
    try {
      return await lambdaClient.send(
        new CreateFunctionCommand({
          Code: {
            ZipFile: Uint8Array.from(readFileSync(lambdaPath)),
          },
          Runtime: Runtime.nodejs18x,
          Handler: "index.handler",
          Layers: [layerVersionArn],
          FunctionName: name,
          Role: roleArn,
        }),
      );
    } catch (caught) {
      if (
        caught instanceof Error &&

```

```

        caught.name === "ResourceConflictException"
    ) {
        const { Configuration } = await lambdaClient.send(
            new GetFunctionCommand({ FunctionName: name }),
        );
        return Configuration;
    } else {
        throw caught;
    }
}
};

// Function creation fails if the Role is not ready. This retries
// function creation until it succeeds or it times out.
const { FunctionArn } = await retry(
    { intervalInMs: 1000, maxRetries: 60 },
    createFunction,
);

return {
    arn: FunctionArn,
    cleanUp: async () => {
        await lambdaClient.send(
            new DeleteFunctionCommand({ FunctionName: name }),
        );
    },
};
}

/**
 * This uploads some sample coordinate data to an Amazon S3 bucket.
 * The Amazon SageMaker Geospatial vector enrichment job will take the simple Lat/
Long
 * coordinates in this file and augment them with more detailed location data.
 * @param {{bucketName: string, s3Client: import('@aws-sdk/client-s3').S3Client}}
props
 */
export async function uploadCSVDataToS3({ bucketName, s3Client }) {
    const s3Path = `${dirnameFromMetaUrl(
        import.meta.url,
    )}../../../../../../../../workflows/sagemaker_pipelines/resources/latlongtest.csv`;

    await s3Client.send(
        new PutObjectCommand({

```

```

        Bucket: bucketName,
        Key: "input/sample_data.csv",
        Body: readFileSync(s3Path),
    })),
    );
}

/**
 * Create the AWS IAM role that will be assumed by the Amazon SageMaker pipeline.
 * @param {{name: string, iamClient: import('@aws-sdk/client-iam').IAMClient, wait:
 * (ms: number) => Promise<void>}} props
 */
export async function createSagemakerRole({ name, iamClient, wait }) {
    let role = null;

    const createRole = () =>
        iamClient.send(
            new CreateRoleCommand({
                RoleName: name,
                AssumeRolePolicyDocument: JSON.stringify({
                    Version: "2012-10-17",
                    Statement: [
                        {
                            Effect: "Allow",
                            Action: ["sts:AssumeRole"],
                            Principal: {
                                Service: [
                                    "sagemaker.amazonaws.com",
                                    "sagemaker-geospatial.amazonaws.com",
                                ],
                            },
                        },
                    ],
                }),
            }),
        );

    try {
        const { Role } = await createRole();
        role = Role;
        // Wait for the role to be ready.
        await wait(10);
    } catch (caught) {
        if (

```

```
        caught instanceof Error &&
        caught.name === "EntityAlreadyExistsException"
    ) {
        const { Role } = await iamClient.send(
            new GetRoleCommand({ RoleName: name }),
        );
        role = Role;
    } else {
        throw caught;
    }
}

return {
    arn: role.Arn,
    cleanUp: async () => {
        await iamClient.send(new DeleteRoleCommand({ RoleName: name }));
    },
};
}

/**
 * Create the Amazon SageMaker execution policy. This policy grants permission to
 * invoke the AWS Lambda function, read/write to the Amazon S3 bucket, and send
 * messages to
 * the Amazon SQS queue.
 * @param {{ name: string, sqsQueueArn: string, lambdaArn: string, iamClient:
 * import('@aws-sdk/client-iam').IAMClient, s3BucketName: string}} props
 */
export async function createSagemakerExecutionPolicy({
    sqsQueueArn,
    lambdaArn,
    iamClient,
    name,
    s3BucketName,
}) {
    const policyConfig = {
        Version: "2012-10-17",
        Statement: [
            {
                Effect: "Allow",
                Action: ["lambda:InvokeFunction"],
                Resource: lambdaArn,
            },
            {

```

```
    Effect: "Allow",
    Action: ["s3:*"],
    Resource: [
      `arn:aws:s3:::${s3BucketName}`,
      `arn:aws:s3:::${s3BucketName}/*`,
    ],
  },
  {
    Effect: "Allow",
    Action: ["sqs:SendMessage"],
    Resource: sqsQueueArn,
  },
],
};

const createPolicy = () =>
  iamClient.send(
    new CreatePolicyCommand({
      PolicyDocument: JSON.stringify(policyConfig),
      PolicyName: name,
    }),
  );

let policy = null;

try {
  const { Policy } = await createPolicy();
  policy = Policy;
} catch (caught) {
  if (
    caught instanceof Error &&
    caught.name === "EntityAlreadyExistsException"
  ) {
    const { Policies } = await iamClient.send(new ListPoliciesCommand({}));
    if (Policies) {
      policy = Policies.find((p) => p.PolicyName === name);
    } else {
      throw new Error("No policies found.");
    }
  } else {
    throw caught;
  }
}
```

```
    return {
      arn: policy?.Arn,
      policyConfig,
      cleanUp: async () => {
        await iamClient.send(new DeletePolicyCommand({ PolicyArn: policy?.Arn }));
      },
    };
  }
}

/**
 * Create the Amazon SageMaker pipeline using a JSON pipeline definition. The
 * definition
 * can also be provided as an Amazon S3 object using PipelineDefinitionS3Location.
 * @param {{roleArn: string, name: string, sagemakerClient: import('@aws-sdk/client-
 * sagemaker').SageMakerClient}} props
 */
export async function createSagemakerPipeline({
  // Assumes an AWS IAM role has been created for this pipeline.
  roleArn,
  name,
  // Assumes an AWS Lambda function has been created for this pipeline.
  functionArn,
  sagemakerClient,
}) {
  const pipelineDefinition = readFileSync(
    // dirnameFromMetaUrl is a local utility function. You can find its
    // implementation
    // on GitHub.
    `${dirnameFromMetaUrl(
      import.meta.url,
    )}../../../../workflows/sagemaker_pipelines/resources/
    GeoSpatialPipeline.json`,
  )
  .toString()
  .replace(/\*FUNCTION_ARN\*/g, functionArn);

  let arn = null;

  const createPipeline = () =>
    sagemakerClient.send(
      new CreatePipelineCommand({
        PipelineName: name,
        PipelineDefinition: pipelineDefinition,
        RoleArn: roleArn,
      })
    );
}
```



```

    }},
  );

  try {
    const { PipelineArn } = await createPipeline();
    arn = PipelineArn;
  } catch (caught) {
    if (
      caught instanceof Error &&
      caught.name === "ValidationException" &&
      caught.message.includes(
        "Pipeline names must be unique within an AWS account and region",
      )
    ) {
      const { PipelineArn } = await sagemakerClient.send(
        new DescribePipelineCommand({ PipelineName: name }),
      );
      arn = PipelineArn;
    } else {
      throw caught;
    }
  }

  return {
    arn,
    cleanUp: async () => {
      await sagemakerClient.send(
        new DeletePipelineCommand({ PipelineName: name }),
      );
    },
  };
}

/**
 * Create an Amazon SQS queue. The Amazon SageMaker pipeline will send messages
 * to this queue that are then processed by the AWS Lambda function.
 * @param {{name: string, sqsClient: import('@aws-sdk/client-sqs').SQSClient}} props
 */
export async function createSQSQueue({ name, sqsClient }) {
  const createSqsQueue = () =>
    sqsClient.send(
      new CreateQueueCommand({
        QueueName: name,
        Attributes: {

```

```
        DelaySeconds: "5",
        ReceiveMessageWaitTimeSeconds: "5",
        VisibilityTimeout: "300",
    },
 )),
);

let queueUrl = null;
try {
  const { QueueUrl } = await createSqsQueue();
  queueUrl = QueueUrl;
} catch (caught) {
  if (caught instanceof Error && caught.name === "QueueNameExists") {
    const { QueueUrl } = await sqsClient.send(
      new GetQueueUrlCommand({ QueueName: name }),
    );
    queueUrl = QueueUrl;
  } else {
    throw caught;
  }
}

const { Attributes } = await retry(
  { intervalInMs: 1000, maxRetries: 60 },
  () =>
    sqsClient.send(
      new GetQueueAttributesCommand({
        QueueUrl: queueUrl,
        AttributeNames: ["QueueArn"],
      }),
    ),
);

return {
  queueUrl,
  queueArn: Attributes.QueueArn,
  cleanUp: async () => {
    await sqsClient.send(new DeleteQueueCommand({ QueueUrl: queueUrl }));
  },
};
}

/**
 * Configure the AWS Lambda function to long poll for messages from the Amazon SQS
```

```

* queue.
* @param {{
*   paginateListEventSourceMappings: () => Generator<import('@aws-sdk/client-
lambda').ListEventSourceMappingsCommandOutput>,
*   lambdaName: string,
*   queueArn: string,
*   lambdaClient: import('@aws-sdk/client-lambda').LambdaClient}} props
*/
export async function configureLambdaSQSEventSource({
  lambdaName,
  queueArn,
  lambdaClient,
  paginateListEventSourceMappings,
}) {
  let uuid = null;
  const createEvenSourceMapping = () =>
    lambdaClient.send(
      new CreateEventSourceMappingCommand({
        EventSourceArn: queueArn,
        FunctionName: lambdaName,
      }),
    );

  try {
    const { UUID } = await createEvenSourceMapping();
    uuid = UUID;
  } catch (caught) {
    if (
      caught instanceof Error &&
      caught.name === "ResourceConflictException"
    ) {
      const paginator = paginateListEventSourceMappings(
        { client: lambdaClient },
        {},
      );
      /**
       * @type {import('@aws-sdk/client-lambda').EventSourceMappingConfiguration[]}
       */
      const eventSourceMappings = [];
      for await (const page of paginator) {
        eventSourceMappings.concat(page.EventSourceMappings || []);
      }

      const { Configuration } = await lambdaClient.send(

```

```

        new GetFunctionCommand({ FunctionName: lambdaName })),
    );

    uuid = eventSourceMappings.find(
        (mapping) =>
            mapping.EventSourceArn === queueArn &&
            mapping.FunctionArn === Configuration.FunctionArn,
    ).UUID;
} else {
    throw caught;
}
}

return {
    cleanUp: async () => {
        await lambdaClient.send(
            new DeleteEventSourceMappingCommand({
                UUID: uuid,
            })),
    );
    },
};
}

/**
 * Create an Amazon S3 bucket that will store the simple coordinate file as input
 * and the output of the Amazon SageMaker Geospatial vector enrichment job.
 * @param {{
 *   s3Client: import('@aws-sdk/client-s3').S3Client,
 *   name: string,
 *   paginateListObjectsV2: () => Generator<import('@aws-sdk/client-
 * s3').ListObjectsCommandOutput>
 * }} props
 */
export async function createS3Bucket({
    name,
    s3Client,
    paginateListObjectsV2,
}) {
    await s3Client.send(new CreateBucketCommand({ Bucket: name }));

    return {
        cleanUp: async () => {
            const paginator = paginateListObjectsV2(

```

```

        { client: s3Client },
        { Bucket: name },
    );
    for await (const page of paginator) {
        const objects = page.Contents;
        if (objects) {
            for (const object of objects) {
                await s3Client.send(
                    new DeleteObjectCommand({ Bucket: name, Key: object.Key }),
                );
            }
        }
    }
    await s3Client.send(new DeleteBucketCommand({ Bucket: name }));
},
};
}

/**
 * Start the execution of the Amazon SageMaker pipeline. Parameters that are
 * passed in are used in the AWS Lambda function.
 * @param {{
 *   name: string,
 *   sagemakerClient: import('@aws-sdk/client-sagemaker').SageMakerClient,
 *   roleArn: string,
 *   queueUrl: string,
 *   s3InputBucketName: string,
 * }} props
 */
export async function startPipelineExecution({
    sagemakerClient,
    name,
    bucketName,
    roleArn,
    queueUrl,
}) {
    /**
     * The Vector Enrichment Job requests CSV data. This configuration points to a CSV
     * file in an Amazon S3 bucket.
     * @type {import("@aws-sdk/client-sagemaker-geospatial").VectorEnrichmentJobInputConfig}
     */
    const inputConfig = {
        DataSourceConfig: {

```

```
    S3Data: {
      S3Uri: `s3://${bucketName}/input/sample_data.csv`,
    },
  },
  DocumentType: VectorEnrichmentJobDocumentType.CSV,
};

/**
 * The Vector Enrichment Job adds additional data to the source CSV. This
 * configuration points
 * to an Amazon S3 prefix where the output will be stored.
 * @type {import("@aws-sdk/client-sagemaker-geospatial").ExportVectorEnrichmentJobOutputConfig}
 */
const outputConfig = {
  S3Data: {
    S3Uri: `s3://${bucketName}/output/`,
  },
};

/**
 * This job will be a Reverse Geocoding Vector Enrichment Job. Reverse Geocoding
 * requires
 * latitude and longitude values.
 * @type {import("@aws-sdk/client-sagemaker-geospatial").VectorEnrichmentJobConfig}
 */
const jobConfig = {
  ReverseGeocodingConfig: {
    XAttributeName: "Longitude",
    YAttributeName: "Latitude",
  },
};

const { PipelineExecutionArn } = await sagemakerClient.send(
  new StartPipelineExecutionCommand({
    PipelineName: name,
    PipelineExecutionDisplayName: `${name}-example-execution`,
    PipelineParameters: [
      { Name: "parameter_execution_role", Value: roleArn },
      { Name: "parameter_queue_url", Value: queueUrl },
      {
        Name: "parameter_vej_input_config",
        Value: JSON.stringify(inputConfig),
      }
    ]
  })
);
```

```

    },
    {
      Name: "parameter_vej_export_config",
      Value: JSON.stringify(outputConfig),
    },
    {
      Name: "parameter_step_1_vej_config",
      Value: JSON.stringify(jobConfig),
    },
  ],
 )),
);

return {
  arn: PipelineExecutionArn,
};
}

/**
 * Poll the executing pipeline until the status is 'SUCCEEDED', 'STOPPED', or
 * 'FAILED'.
 * @param {{ arn: string, sagemakerClient: import('@aws-sdk/client-
 * sagemaker').SageMakerClient, wait: (ms: number) => Promise<void>}} props
 */
export async function waitForPipelineComplete({ arn, sagemakerClient, wait }) {
  const command = new DescribePipelineExecutionCommand({
    PipelineExecutionArn: arn,
  });

  let complete = false;
  let intervalInSeconds = 15;
  const COMPLETION_STATUSES = [
    PipelineExecutionStatus.FAILED,
    PipelineExecutionStatus.STOPPED,
    PipelineExecutionStatus.SUCCEEDED,
  ];

  do {
    const { PipelineExecutionStatus: status, FailureReason } =
      await sagemakerClient.send(command);

    complete = COMPLETION_STATUSES.includes(status);

    if (!complete) {

```

```
    console.log(
      `Pipeline is ${status}. Waiting ${intervalInSeconds} seconds before checking
again.`,
    );
    await wait(intervalInSeconds);
  } else if (status === PipelineExecutionStatus.FAILED) {
    throw new Error(`Pipeline failed because: ${FailureReason}`);
  } else if (status === PipelineExecutionStatus.STOPPED) {
    throw new Error(`Pipeline was forcefully stopped.`);
  } else {
    console.log(`Pipeline execution ${status}.`);
  }
} while (!complete);
}

/**
 * Return the string value of an Amazon S3 object.
 * @param {{ bucket: string, key: string, s3Client: import('@aws-sdk/client-
s3').S3Client}} param0
 */
export async function getObject({ bucket, s3Client }) {
  const prefix = "output/";
  const { Contents } = await s3Client.send(
    new ListObjectsV2Command({ MaxKeys: 1, Bucket: bucket, Prefix: prefix }),
  );

  if (!Contents.length) {
    throw new Error("No objects found in bucket.");
  }

  // Find the CSV file.
  const outputObject = Contents.find((obj) => obj.Key.endsWith(".csv"));

  if (!outputObject) {
    throw new Error(`No CSV file found in bucket with the prefix "${prefix}.`);
  }

  const { Body } = await s3Client.send(
    new GetObjectCommand({
      Bucket: bucket,
      Key: outputObject.Key,
    }),
  );
};
```



```
    return Body.transformToString();
  }
```

This function is an excerpt from a file that uses the preceding library functions to set up a SageMaker pipeline, execute it, and delete all created resources.

```
import { retry, wait } from "@aws-doc-sdk-examples/lib/utils/util-timers.js";
import {
  attachPolicy,
  configureLambdaSQSEventSource,
  createLambdaExecutionPolicy,
  createLambdaExecutionRole,
  createLambdaFunction,
  createLambdaLayer,
  createS3Bucket,
  createSQSQueue,
  createSagemakerExecutionPolicy,
  createSagemakerPipeline,
  createSagemakerRole,
  getObject,
  startPipelineExecution,
  uploadCSVDataToS3,
  waitForPipelineComplete,
} from "./lib.js";
import { MESSAGES } from "./messages.js";

export class SageMakerPipelinesWkflw {
  names = {
    LAMBDA_EXECUTION_ROLE: "sagemaker-wkflw-lambda-execution-role",
    LAMBDA_EXECUTION_ROLE_POLICY:
      "sagemaker-wkflw-lambda-execution-role-policy",
    LAMBDA_FUNCTION: "sagemaker-wkflw-lambda-function",
    LAMBDA_LAYER: "sagemaker-wkflw-lambda-layer",
    SAGE_MAKER_EXECUTION_ROLE: "sagemaker-wkflw-pipeline-execution-role",
    SAGE_MAKER_EXECUTION_ROLE_POLICY:
      "sagemaker-wkflw-pipeline-execution-role-policy",
    SAGE_MAKER_PIPELINE: "sagemaker-wkflw-pipeline",
    SQS_QUEUE: "sagemaker-wkflw-sqs-queue",
    S3_BUCKET: `sagemaker-wkflw-s3-bucket-${Date.now()}`,
  };

  cleanUpFunctions = [];
```

```
/**
 * @param {import("@aws-doc-sdk-examples/lib/prompter.js").Prompter} prompter
 * @param {import("@aws-doc-sdk-examples/lib/logger.js").Logger} logger
 * @param {{ IAM: import("@aws-sdk/client-iam").IAMClient, Lambda: import("@aws-
sdk/client-lambda").LambdaClient, SageMaker: import("@aws-sdk/client-
sagemaker").SageMakerClient, S3: import("@aws-sdk/client-s3").S3Client, SQS:
import("@aws-sdk/client-sqs").SQSClient }} clients
 */
constructor(prompter, logger, clients) {
  this.prompter = prompter;
  this.logger = logger;
  this.clients = clients;
}

async run() {
  try {
    await this.startWorkflow();
  } catch (err) {
    console.error(err);
    throw err;
  } finally {
    this.logger.logSeparator();
    const doCleanUp = await this.prompter.confirm({
      message: "Clean up resources?",
    });
    if (doCleanUp) {
      await this.cleanUp();
    }
  }
}

async cleanUp() {
  // Run all of the clean up functions. If any fail, we log the error and
  continue.
  // This ensures all clean up functions are run.
  for (let i = this.cleanUpFunctions.length - 1; i >= 0; i--) {
    await retry(
      { intervalInMs: 1000, maxRetries: 60, swallowError: true },
      this.cleanUpFunctions[i],
    );
  }
}
```

```
async startWorkflow() {
  this.logger.logSeparator(MESSAGES.greetingHeader);
  await this.logger.log(MESSAGES.greeting);

  this.logger.logSeparator();
  await this.logger.log(
    MESSAGES.creatingRole.replace(
      "${ROLE_NAME}",
      this.names.LAMBDA_EXECUTION_ROLE,
    ),
  );

  // Create an IAM role that will be assumed by the AWS Lambda function. This
  // function
  // is triggered by Amazon SQS messages and calls SageMaker and SageMaker
  // GeoSpatial actions.
  const { arn: lambdaExecutionRoleArn, cleanUp: lambdaExecutionRoleCleanUp } =
    await createLambdaExecutionRole({
      name: this.names.LAMBDA_EXECUTION_ROLE,
      iamClient: this.clients.IAM,
    });
  // Add a clean up step to a stack for every resource created.
  this.cleanUpFunctions.push(lambdaExecutionRoleCleanUp);

  await this.logger.log(
    MESSAGES.roleCreated.replace(
      "${ROLE_NAME}",
      this.names.LAMBDA_EXECUTION_ROLE,
    ),
  );

  this.logger.logSeparator();

  await this.logger.log(
    MESSAGES.creatingRole.replace(
      "${ROLE_NAME}",
      this.names.SAGE_MAKER_EXECUTION_ROLE,
    ),
  );

  // Create an IAM role that will be assumed by the SageMaker pipeline. The
  // pipeline
  // sends messages to an Amazon SQS queue and puts/retrieves Amazon S3 objects.
  const {
```

```
    arn: pipelineExecutionRoleArn,
    cleanUp: pipelineExecutionRoleCleanUp,
  } = await createSagemakerRole({
    iamClient: this.clients.IAM,
    name: this.names.SAGE_MAKER_EXECUTION_ROLE,
    wait,
  });
  this.cleanUpFunctions.push(pipelineExecutionRoleCleanUp);

  await this.logger.log(
    MESSAGES.roleCreated.replace(
      "${ROLE_NAME}",
      this.names.SAGE_MAKER_EXECUTION_ROLE,
    ),
  );

  this.logger.logSeparator();

  // Create an IAM policy that allows the AWS Lambda function to invoke SageMaker
  APIs.
  const {
    arn: lambdaExecutionPolicyArn,
    policy: lambdaPolicy,
    cleanUp: lambdaExecutionPolicyCleanUp,
  } = await createLambdaExecutionPolicy({
    name: this.names.LAMBDA_EXECUTION_ROLE_POLICY,
    s3BucketName: this.names.S3_BUCKET,
    iamClient: this.clients.IAM,
    pipelineExecutionRoleArn,
  });
  this.cleanUpFunctions.push(lambdaExecutionPolicyCleanUp);

  console.log(JSON.stringify(lambdaPolicy, null, 2), "\n");

  await this.logger.log(
    MESSAGES.attachPolicy
      .replace("${POLICY_NAME}", this.names.LAMBDA_EXECUTION_ROLE_POLICY)
      .replace("${ROLE_NAME}", this.names.LAMBDA_EXECUTION_ROLE),
  );

  await this.prompter.checkContinue();

  // Attach the Lambda execution policy to the execution role.
  const { cleanUp: lambdaExecutionRolePolicyCleanUp } = await attachPolicy({
```

```
    roleName: this.names.LAMBDA_EXECUTION_ROLE,
    policyArn: lambdaExecutionPolicyArn,
    iamClient: this.clients.IAM,
  });
  this.cleanupFunctions.push(lambdaExecutionRolePolicyCleanup);

  await this.logger.log(MESSAGES.policyAttached);

  this.logger.logSeparator();

  // Create Lambda layer for SageMaker packages.
  const { versionArn: layerVersionArn, cleanup: lambdaLayerCleanup } =
    await createLambdaLayer({
      name: this.names.LAMBDA_LAYER,
      lambdaClient: this.clients.Lambda,
    });
  this.cleanupFunctions.push(lambdaLayerCleanup);

  await this.logger.log(
    MESSAGES.creatingFunction.replace(
      "${FUNCTION_NAME}",
      this.names.LAMBDA_FUNCTION,
    ),
  );

  // Create the Lambda function with the execution role.
  const { arn: lambdaArn, cleanup: lambdaCleanup } =
    await createLambdaFunction({
      roleArn: lambdaExecutionRoleArn,
      lambdaClient: this.clients.Lambda,
      name: this.names.LAMBDA_FUNCTION,
      layerVersionArn,
    });
  this.cleanupFunctions.push(lambdaCleanup);

  await this.logger.log(
    MESSAGES.functionCreated.replace(
      "${FUNCTION_NAME}",
      this.names.LAMBDA_FUNCTION,
    ),
  );

  this.logger.logSeparator();
```

```
await this.logger.log(
  MESSAGES.creatingSQSQueue.replace("${QUEUE_NAME}", this.names.SQS_QUEUE),
);

// Create an SQS queue for the SageMaker pipeline.
const {
  queueUrl,
  queueArn,
  cleanUp: queueCleanUp,
} = await createSQSQueue({
  name: this.names.SQS_QUEUE,
  sqsClient: this.clients.SQS,
});
this.cleanUpFunctions.push(queueCleanUp);

await this.logger.log(
  MESSAGES.sqsQueueCreated.replace("${QUEUE_NAME}", this.names.SQS_QUEUE),
);

this.logger.logSeparator();

await this.logger.log(
  MESSAGES.configuringLambdaSQSEventSource
    .replace("${LAMBDA_NAME}", this.names.LAMBDA_FUNCTION)
    .replace("${QUEUE_NAME}", this.names.SQS_QUEUE),
);

// Configure the SQS queue as an event source for the Lambda.
const { cleanUp: lambdaSQSEventSourceCleanUp } =
  await configureLambdaSQSEventSource({
    lambdaArn,
    lambdaName: this.names.LAMBDA_FUNCTION,
    queueArn,
    sqsClient: this.clients.SQS,
    lambdaClient: this.clients.Lambda,
  });
this.cleanUpFunctions.push(lambdaSQSEventSourceCleanUp);

await this.logger.log(
  MESSAGES.lambdaSQSEventSourceConfigured
    .replace("${LAMBDA_NAME}", this.names.LAMBDA_FUNCTION)
    .replace("${QUEUE_NAME}", this.names.SQS_QUEUE),
);
```

```
this.logger.logSeparator();

// Create an IAM policy that allows the SageMaker pipeline to invoke AWS Lambda
// and send messages to the Amazon SQS queue.
const {
  arn: pipelineExecutionPolicyArn,
  policy: sagemakerPolicy,
  cleanUp: pipelineExecutionPolicyCleanUp,
} = await createSagemakerExecutionPolicy({
  sqsQueueArn: queueArn,
  lambdaArn,
  iamClient: this.clients.IAM,
  name: this.names.SAGE_MAKER_EXECUTION_ROLE_POLICY,
  s3BucketName: this.names.S3_BUCKET,
});
this.cleanUpFunctions.push(pipelineExecutionPolicyCleanUp);

console.log(JSON.stringify(sagemakerPolicy, null, 2));

await this.logger.log(
  MESSAGES.attachPolicy
    .replace("${POLICY_NAME}", this.names.SAGE_MAKER_EXECUTION_ROLE_POLICY)
    .replace("${ROLE_NAME}", this.names.SAGE_MAKER_EXECUTION_ROLE),
);

await this.prompter.checkContinue();

// Attach the SageMaker execution policy to the execution role.
const { cleanUp: pipelineExecutionRolePolicyCleanUp } = await attachPolicy({
  roleName: this.names.SAGE_MAKER_EXECUTION_ROLE,
  policyArn: pipelineExecutionPolicyArn,
  iamClient: this.clients.IAM,
});
this.cleanUpFunctions.push(pipelineExecutionRolePolicyCleanUp);
// Wait for the role to be ready. If the role is used immediately,
// the pipeline will fail.
await wait(5);

await this.logger.log(MESSAGES.policyAttached);

this.logger.logSeparator();

await this.logger.log(
  MESSAGES.creatingPipeline.replace(
```

```
        "${PIPELINE_NAME}",
        this.names.SAGE_MAKER_PIPELINE,
    ),
);

// Create the SageMaker pipeline.
const { cleanUp: pipelineCleanUp } = await createSagemakerPipeline({
    roleArn: pipelineExecutionRoleArn,
    functionArn: lambdaArn,
    sagemakerClient: this.clients.SageMaker,
    name: this.names.SAGE_MAKER_PIPELINE,
});
this.cleanUpFunctions.push(pipelineCleanUp);

await this.logger.log(
    MESSAGES.pipelineCreated.replace(
        "${PIPELINE_NAME}",
        this.names.SAGE_MAKER_PIPELINE,
    ),
);

this.logger.logSeparator();

await this.logger.log(
    MESSAGES.creatingS3Bucket.replace("${BUCKET_NAME}", this.names.S3_BUCKET),
);

// Create an S3 bucket for storing inputs and outputs.
const { cleanUp: s3BucketCleanUp } = await createS3Bucket({
    name: this.names.S3_BUCKET,
    s3Client: this.clients.S3,
});
this.cleanUpFunctions.push(s3BucketCleanUp);

await this.logger.log(
    MESSAGES.s3BucketCreated.replace("${BUCKET_NAME}", this.names.S3_BUCKET),
);

this.logger.logSeparator();

await this.logger.log(
    MESSAGES.uploadingInputData.replace(
        "${BUCKET_NAME}",
        this.names.S3_BUCKET,
```



```
    ),
  );

  // Upload CSV Lat/Long data to S3.
  await uploadCSVDataToS3({
    bucketName: this.names.S3_BUCKET,
    s3Client: this.clients.S3,
  });

  await this.logger.log(MESSAGES.inputDataUploaded);

  this.logger.logSeparator();

  await this.prompter.checkContinue(MESSAGES.executePipeline);

  // Execute the SageMaker pipeline.
  const { arn: pipelineExecutionArn } = await startPipelineExecution({
    name: this.names.SAGE_MAKER_PIPELINE,
    sagemakerClient: this.clients.SageMaker,
    roleArn: pipelineExecutionRoleArn,
    bucketName: this.names.S3_BUCKET,
    queueUrl,
  });

  // Wait for the pipeline execution to finish.
  await waitForPipelineComplete({
    arn: pipelineExecutionArn,
    sagemakerClient: this.clients.SageMaker,
    wait,
  });

  this.logger.logSeparator();

  await this.logger.log(MESSAGES.outputDelay);

  // The getOutput function will throw an error if the output is not
  // found. The retry function will retry a failed function call once
  // ever 10 seconds for 2 minutes.
  const output = await retry({ intervalInMs: 10000, maxRetries: 12 }, () =>
    getObject({
      bucket: this.names.S3_BUCKET,
      s3Client: this.clients.S3,
    })),
  );
```

```
    this.logger.logSeparator();
    await this.logger.log(MESSAGES.outputDataRetrieved);
    console.log(output.split("\n").slice(0, 6).join("\n"));
  }
}
```

- For API details, see the following topics in *AWS SDK for JavaScript API Reference*.
 - [CreatePipeline](#)
 - [DeletePipeline](#)
 - [DescribePipelineExecution](#)
 - [StartPipelineExecution](#)
 - [UpdatePipeline](#)

Secrets Manager examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with Secrets Manager.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Topics

- [Actions](#)

Actions

GetSecretValue

The following code example shows how to use `GetSecretValue`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import {
  GetSecretValueCommand,
  SecretsManagerClient,
} from "@aws-sdk/client-secrets-manager";

export const getSecretValue = async (secretName = "SECRET_NAME") => {
  const client = new SecretsManagerClient();
  const response = await client.send(
    new GetSecretValueCommand({
      SecretId: secretName,
    }),
  );
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: '584eb612-f8b0-48c9-855e-6d246461b604',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  //   ARN: 'arn:aws:secretsmanager:us-east-1:xxxxxxxxxxxx:secret:binary-
secret-3873048-xxxxxx',
  //   CreatedDate: 2023-08-08T19:29:51.294Z,
  //   Name: 'binary-secret-3873048',
  //   SecretBinary: Uint8Array(11) [
  //     98, 105, 110, 97, 114,
  //     121, 32, 100, 97, 116,
  //     97
  //   ],
  //   VersionId: '712083f4-0d26-415e-8044-16735142cd6a',
  //   VersionStages: [ 'AWSCURRENT' ]
  // }
```

```
if (response.SecretString) {
    return response.SecretString;
}

if (response.SecretBinary) {
    return response.SecretBinary;
}
};
```

- For API details, see [GetSecretValue](#) in *AWS SDK for JavaScript API Reference*.

Amazon SES examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with Amazon SES.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Scenarios are code examples that show you how to accomplish specific tasks by calling multiple functions within a service or combined with other AWS services.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Topics

- [Actions](#)
- [Scenarios](#)

Actions

CreateReceiptFilter

The following code example shows how to use `CreateReceiptFilter`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import {
  CreateReceiptFilterCommand,
  ReceiptFilterPolicy,
} from "@aws-sdk/client-ses";
import { sesClient } from "../libs/sesClient.js";
import { getUniqueName } from "@aws-doc-sdk-examples/lib/utils/util-string.js";

const createCreateReceiptFilterCommand = ({ policy, ipOrRange, name }) => {
  return new CreateReceiptFilterCommand({
    Filter: {
      IpFilter: {
        Cidr: ipOrRange, // string, either a single IP address (10.0.0.1) or an IP
        address range in CIDR notation (10.0.0.1/24)).
        Policy: policy, // enum ReceiptFilterPolicy, email traffic from the filtered
        addressesOptions.
      },
      /*
       * The name of the IP address filter. Only ASCII letters, numbers, underscores,
       * or dashes.
       * Must be less than 64 characters and start and end with a letter or number.
       */
      Name: name,
    },
  });
};

const FILTER_NAME = getUniqueName("ReceiptFilter");

const run = async () => {
  const createReceiptFilterCommand = createCreateReceiptFilterCommand({
    policy: ReceiptFilterPolicy.Allow,
    ipOrRange: "10.0.0.1",
    name: FILTER_NAME,
  });
};
```

```
try {
  return await sesClient.send(createReceiptFilterCommand);
} catch (caught) {
  if (caught instanceof Error && caught.name === "MessageRejected") {
    /** @type { import('@aws-sdk/client-ses').MessageRejected } */
    const messageRejectedError = caught;
    return messageRejectedError;
  }
  throw caught;
}
};
```

- For API details, see [CreateReceiptFilter](#) in *AWS SDK for JavaScript API Reference*.

CreateReceiptRule

The following code example shows how to use `CreateReceiptRule`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { CreateReceiptRuleCommand, TlsPolicy } from "@aws-sdk/client-ses";
import { sesClient } from "../libs/sesClient.js";
import { getUniqueName } from "@aws-doc-sdk-examples/lib/utils/util-string.js";

const RULE_SET_NAME = getUniqueName("RuleSetName");
const RULE_NAME = getUniqueName("RuleName");
const S3_BUCKET_NAME = getUniqueName("S3BucketName");

const createS3ReceiptRuleCommand = ({
  bucketName,
  emailAddresses,
  name,
  ruleSet,
```

```
    }) => {
      return new CreateReceiptRuleCommand({
        Rule: {
          Actions: [
            {
              S3Action: {
                BucketName: bucketName,
                ObjectKeyPrefix: "email",
              },
            },
          ],
          Recipients: emailAddresses,
          Enabled: true,
          Name: name,
          ScanEnabled: false,
          TlsPolicy: TlsPolicy.Optional,
        },
        RuleSetName: ruleSet, // Required
      });
    };

const run = async () => {
  const s3ReceiptRuleCommand = createS3ReceiptRuleCommand({
    bucketName: S3_BUCKET_NAME,
    emailAddresses: ["email@example.com"],
    name: RULE_NAME,
    ruleSet: RULE_SET_NAME,
  });

  try {
    return await sesClient.send(s3ReceiptRuleCommand);
  } catch (err) {
    console.log("Failed to create S3 receipt rule.", err);
    throw err;
  }
};
```

- For API details, see [CreateReceiptRule](#) in *AWS SDK for JavaScript API Reference*.

CreateReceiptRuleSet

The following code example shows how to use CreateReceiptRuleSet.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { CreateReceiptRuleSetCommand } from "@aws-sdk/client-ses";
import { sesClient } from "../libs/sesClient.js";
import { getUniqueName } from "@aws-doc-sdk-examples/lib/utils/util-string.js";

const RULE_SET_NAME = getUniqueName("RuleSetName");

const createCreateReceiptRuleSetCommand = (ruleSetName) => {
  return new CreateReceiptRuleSetCommand({ RuleSetName: ruleSetName });
};

const run = async () => {
  const createReceiptRuleSetCommand =
    createCreateReceiptRuleSetCommand(RULE_SET_NAME);

  try {
    return await sesClient.send(createReceiptRuleSetCommand);
  } catch (err) {
    console.log("Failed to create receipt rule set", err);
    return err;
  }
};
```

- For API details, see [CreateReceiptRuleSet](#) in *AWS SDK for JavaScript API Reference*.

CreateTemplate

The following code example shows how to use CreateTemplate.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { CreateTemplateCommand } from "@aws-sdk/client-ses";
import { sesClient } from "../libs/sesClient.js";
import { getUniqueName } from "@aws-doc-sdk-examples/lib/utils/util-string.js";

const TEMPLATE_NAME = getUniqueName("TestTemplateName");

const createCreateTemplateCommand = () => {
  return new CreateTemplateCommand({
    /**
     * The template feature in Amazon SES is based on the Handlebars template
     system.
     */
    Template: {
      /**
       * The name of an existing template in Amazon SES.
       */
      TemplateName: TEMPLATE_NAME,
      HtmlPart: `
        <h1>Hello, {{contact.firstName}}!</h1>
        <p>
          Did you know Amazon has a mascot named Peccy?
        </p>
      `,
      SubjectPart: "Amazon Tip",
    },
  });
};

const run = async () => {
  const createTemplateCommand = createCreateTemplateCommand();

  try {
    return await sesClient.send(createTemplateCommand);
  }
};
```

```
    } catch (err) {
      console.log("Failed to create template.", err);
      return err;
    }
  };
```

- For API details, see [CreateTemplate](#) in *AWS SDK for JavaScript API Reference*.

DeleteIdentity

The following code example shows how to use DeleteIdentity.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { DeleteIdentityCommand } from "@aws-sdk/client-ses";
import { sesClient } from "../libs/sesClient.js";

const IDENTITY_EMAIL = "fake@example.com";

const createDeleteIdentityCommand = (identityName) => {
  return new DeleteIdentityCommand({
    Identity: identityName,
  });
};

const run = async () => {
  const deleteIdentityCommand = createDeleteIdentityCommand(IDENTITY_EMAIL);

  try {
    return await sesClient.send(deleteIdentityCommand);
  } catch (err) {
    console.log("Failed to delete identity.", err);
    return err;
  }
}
```

```
};
```

- For API details, see [DeleteIdentity](#) in *AWS SDK for JavaScript API Reference*.

DeleteReceiptFilter

The following code example shows how to use DeleteReceiptFilter.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { DeleteReceiptFilterCommand } from "@aws-sdk/client-ses";
import { sesClient } from "../libs/sesClient.js";
import { getUniqueName } from "@aws-doc-sdk-examples/lib/utils/util-string.js";

const RECEIPT_FILTER_NAME = getUniqueName("ReceiptFilterName");

const createDeleteReceiptFilterCommand = (filterName) => {
  return new DeleteReceiptFilterCommand({ FilterName: filterName });
};

const run = async () => {
  const deleteReceiptFilterCommand =
    createDeleteReceiptFilterCommand(RECEIPT_FILTER_NAME);

  try {
    return await sesClient.send(deleteReceiptFilterCommand);
  } catch (err) {
    console.log("Error deleting receipt filter.", err);
    return err;
  }
};
```

- For API details, see [DeleteReceiptFilter](#) in *AWS SDK for JavaScript API Reference*.

DeleteReceiptRule

The following code example shows how to use DeleteReceiptRule.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { DeleteReceiptRuleCommand } from "@aws-sdk/client-ses";
import { getUniqueName } from "@aws-doc-sdk-examples/lib/utils/util-string.js";
import { sesClient } from "../libs/sesClient.js";

const RULE_NAME = getUniqueName("RuleName");
const RULE_SET_NAME = getUniqueName("RuleSetName");

const createDeleteReceiptRuleCommand = () => {
  return new DeleteReceiptRuleCommand({
    RuleName: RULE_NAME,
    RuleSetName: RULE_SET_NAME,
  });
};

const run = async () => {
  const deleteReceiptRuleCommand = createDeleteReceiptRuleCommand();
  try {
    return await sesClient.send(deleteReceiptRuleCommand);
  } catch (err) {
    console.log("Failed to delete receipt rule.", err);
    return err;
  }
};
```

- For API details, see [DeleteReceiptRule](#) in *AWS SDK for JavaScript API Reference*.

DeleteReceiptRuleSet

The following code example shows how to use DeleteReceiptRuleSet.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { DeleteReceiptRuleSetCommand } from "@aws-sdk/client-ses";
import { getUniqueName } from "@aws-doc-sdk-examples/lib/utils/util-string.js";
import { sesClient } from "../libs/sesClient.js";

const RULE_SET_NAME = getUniqueName("RuleSetName");

const createDeleteReceiptRuleSetCommand = () => {
  return new DeleteReceiptRuleSetCommand({ RuleSetName: RULE_SET_NAME });
};

const run = async () => {
  const deleteReceiptRuleSetCommand = createDeleteReceiptRuleSetCommand();

  try {
    return await sesClient.send(deleteReceiptRuleSetCommand);
  } catch (err) {
    console.log("Failed to delete receipt rule set.", err);
    return err;
  }
};
```

- For API details, see [DeleteReceiptRuleSet](#) in *AWS SDK for JavaScript API Reference*.

DeleteTemplate

The following code example shows how to use DeleteTemplate.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { DeleteTemplateCommand } from "@aws-sdk/client-ses";
import { getUniqueName } from "@aws-doc-sdk-examples/lib/utils/util-string.js";
import { sesClient } from "../libs/sesClient.js";

const TEMPLATE_NAME = getUniqueName("TemplateName");

const createDeleteTemplateCommand = (templateName) =>
  new DeleteTemplateCommand({ TemplateName: templateName });

const run = async () => {
  const deleteTemplateCommand = createDeleteTemplateCommand(TEMPLATE_NAME);

  try {
    return await sesClient.send(deleteTemplateCommand);
  } catch (err) {
    console.log("Failed to delete template.", err);
    return err;
  }
};
```

- For API details, see [DeleteTemplate](#) in *AWS SDK for JavaScript API Reference*.

GetTemplate

The following code example shows how to use GetTemplate.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { GetTemplateCommand } from "@aws-sdk/client-ses";
import { getUniqueName } from "@aws-doc-sdk-examples/lib/utils/util-string.js";
import { sesClient } from "../libs/sesClient.js";

const TEMPLATE_NAME = getUniqueName("TemplateName");

const createGetTemplateCommand = (templateName) =>
  new GetTemplateCommand({ TemplateName: templateName });

const run = async () => {
  const getTemplateCommand = createGetTemplateCommand(TEMPLATE_NAME);

  try {
    return await sesClient.send(getTemplateCommand);
  } catch (caught) {
    if (caught instanceof Error && caught.name === "MessageRejected") {
      /** @type { import('@aws-sdk/client-ses').MessageRejected } */
      const messageRejectedError = caught;
      return messageRejectedError;
    }
    throw caught;
  }
};
```

- For API details, see [GetTemplate](#) in *AWS SDK for JavaScript API Reference*.

ListIdentities

The following code example shows how to use `ListIdentities`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { ListIdentitiesCommand } from "@aws-sdk/client-ses";
import { sesClient } from "../libs/sesClient.js";

const createListIdentitiesCommand = () =>
  new ListIdentitiesCommand({ IdentityType: "EmailAddress", MaxItems: 10 });

const run = async () => {
  const listIdentitiesCommand = createListIdentitiesCommand();

  try {
    return await sesClient.send(listIdentitiesCommand);
  } catch (err) {
    console.log("Failed to list identities.", err);
    return err;
  }
};
```

- For API details, see [ListIdentities](#) in *AWS SDK for JavaScript API Reference*.

ListReceiptFilters

The following code example shows how to use ListReceiptFilters.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).


```
import { ListReceiptFiltersCommand } from "@aws-sdk/client-ses";
import { sesClient } from "../libs/sesClient.js";

const createListReceiptFiltersCommand = () => new ListReceiptFiltersCommand({});

const run = async () => {
  const listReceiptFiltersCommand = createListReceiptFiltersCommand();

  return await sesClient.send(listReceiptFiltersCommand);
};
```

- For API details, see [ListReceiptFilters](#) in *AWS SDK for JavaScript API Reference*.

ListTemplates

The following code example shows how to use ListTemplates.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { ListTemplatesCommand } from "@aws-sdk/client-ses";
import { sesClient } from "../libs/sesClient.js";

const createListTemplatesCommand = (maxItems) =>
  new ListTemplatesCommand({ MaxItems: maxItems });

const run = async () => {
  const listTemplatesCommand = createListTemplatesCommand(10);

  try {
    return await sesClient.send(listTemplatesCommand);
  } catch (err) {
    console.log("Failed to list templates.", err);
    return err;
  }
};
```

```
};
```

- For API details, see [ListTemplates](#) in *AWS SDK for JavaScript API Reference*.

SendBulkTemplatedEmail

The following code example shows how to use `SendBulkTemplatedEmail`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { SendBulkTemplatedEmailCommand } from "@aws-sdk/client-ses";
import {
  getUniqueName,
  postfix,
} from "@aws-doc-sdk-examples/lib/utils/util-string.js";
import { sesClient } from "../libs/sesClient.js";

/**
 * Replace this with the name of an existing template.
 */
const TEMPLATE_NAME = getUniqueName("ReminderTemplate");

/**
 * Replace these with existing verified emails.
 */
const VERIFIED_EMAIL_1 = postfix(getUniqueName("Bilbo"), "@example.com");
const VERIFIED_EMAIL_2 = postfix(getUniqueName("Frodo"), "@example.com");

const USERS = [
  { firstName: "Bilbo", emailAddress: VERIFIED_EMAIL_1 },
  { firstName: "Frodo", emailAddress: VERIFIED_EMAIL_2 },
];

/**
 *
```

```

* @param { { emailAddress: string, firstName: string }[] } users
* @param { string } templateName the name of an existing template in SES
* @returns { SendBulkTemplatedEmailCommand }
*/
const createBulkReminderEmailCommand = (users, templateName) => {
  return new SendBulkTemplatedEmailCommand({
    /**
     * Each 'Destination' uses a corresponding set of replacement data. We can map
     each user
     * to a 'Destination' and provide user specific replacement data to create
     personalized emails.
     *
     * Here's an example of how a template would be replaced with user data:
     * Template: <h1>Hello {{name}},</h1><p>Don't forget about the party gifts!</p>
     * Destination 1: <h1>Hello Bilbo,</h1><p>Don't forget about the party gifts!</
p>
     * Destination 2: <h1>Hello Frodo,</h1><p>Don't forget about the party gifts!</
p>
     */
    Destinations: users.map((user) => ({
      Destination: { ToAddresses: [user.emailAddress] },
      ReplacementTemplateData: JSON.stringify({ name: user.firstName }),
    })),
    DefaultTemplateData: JSON.stringify({ name: "Shireling" }),
    Source: VERIFIED_EMAIL_1,
    Template: templateName,
  });
};

const run = async () => {
  const sendBulkTemplateEmailCommand = createBulkReminderEmailCommand(
    USERS,
    TEMPLATE_NAME,
  );
  try {
    return await sesClient.send(sendBulkTemplateEmailCommand);
  } catch (caught) {
    if (caught instanceof Error && caught.name === "MessageRejected") {
      /** @type { import('@aws-sdk/client-ses').MessageRejected } */
      const messageRejectedError = caught;
      return messageRejectedError;
    }
    throw caught;
  }
}

```

```
};
```

- For API details, see [SendBulkTemplatedEmail](#) in *AWS SDK for JavaScript API Reference*.

SendEmail

The following code example shows how to use `SendEmail`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { SendEmailCommand } from "@aws-sdk/client-ses";
import { sesClient } from "../libs/sesClient.js";

const createSendEmailCommand = (toAddress, fromAddress) => {
  return new SendEmailCommand({
    Destination: {
      /* required */
      CcAddresses: [
        /* more items */
      ],
      ToAddresses: [
        toAddress,
        /* more To-email addresses */
      ],
    },
    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: "EMAIL_SUBJECT",
},
},
Source: fromAddress,
ReplyToAddresses: [
    /* more items */
],
});
};

const run = async () => {
    const sendEmailCommand = createSendEmailCommand(
        "recipient@example.com",
        "sender@example.com",
    );

    try {
        return await sesClient.send(sendEmailCommand);
    } catch (caught) {
        if (caught instanceof Error && caught.name === "MessageRejected") {
            /** @type { import('@aws-sdk/client-ses').MessageRejected } */
            const messageRejectedError = caught;
            return messageRejectedError;
        }
        throw caught;
    }
};
```

- For API details, see [SendEmail](#) in *AWS SDK for JavaScript API Reference*.

SendRawEmail

The following code example shows how to use `SendRawEmail`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Use [nodemailer](#) to send an email with an attachment.

```
import sesClientModule from "@aws-sdk/client-ses";
/**
 * nodemailer wraps the SES SDK and calls SendRawEmail. Use this for more advanced
 * functionality like adding attachments to your email.
 *
 * https://nodemailer.com/transports/ses/
 */
import nodemailer from "nodemailer";

/**
 * @param {string} from An Amazon SES verified email address.
 * @param {*} to An Amazon SES verified email address.
 */
export const sendEmailWithAttachments = (
  from = "from@example.com",
  to = "to@example.com",
) => {
  const ses = new sesClientModule.SESClient({});
  const transporter = nodemailer.createTransport({
    SES: { ses, aws: sesClientModule },
  });

  return new Promise((resolve, reject) => {
    transporter.sendMail(
      {
        from,
        to,
        subject: "Hello World",
        text: "Greetings from Amazon SES!",
        attachments: [{ content: "Hello World!", filename: "hello.txt" }],
      },
      (err, info) => {
        if (err) {
```

```
        reject(err);
      } else {
        resolve(info);
      }
    },
  );
});
};
```

- For API details, see [SendRawEmail](#) in *AWS SDK for JavaScript API Reference*.

SendTemplatedEmail

The following code example shows how to use `SendTemplatedEmail`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { SendTemplatedEmailCommand } from "@aws-sdk/client-ses";
import {
  getUniqueName,
  postfix,
} from "@aws-doc-sdk-examples/lib/utils/util-string.js";
import { sesClient } from "../libs/sesClient.js";

/**
 * Replace this with the name of an existing template.
 */
const TEMPLATE_NAME = getUniqueName("ReminderTemplate");

/**
 * Replace these with existing verified emails.
 */
const VERIFIED_EMAIL = postfix(getUniqueName("Bilbo"), "@example.com");
```

```
const USER = { firstName: "Bilbo", emailAddress: VERIFIED_EMAIL };

/**
 *
 * @param { { emailAddress: string, firstName: string } } user
 * @param { string } templateName - The name of an existing template in Amazon SES.
 * @returns { SendTemplatedEmailCommand }
 */
const createReminderEmailCommand = (user, templateName) => {
  return new SendTemplatedEmailCommand({
    /**
     * Here's an example of how a template would be replaced with user data:
     * Template: <h1>Hello {{contact.firstName}},</h1><p>Don't forget about the
party gifts!</p>
     * Destination: <h1>Hello Bilbo,</h1><p>Don't forget about the party gifts!</p>
     */
    Destination: { ToAddresses: [user.emailAddress] },
    TemplateData: JSON.stringify({ contact: { firstName: user.firstName } }),
    Source: VERIFIED_EMAIL,
    Template: templateName,
  });
};

const run = async () => {
  const sendReminderEmailCommand = createReminderEmailCommand(
    USER,
    TEMPLATE_NAME,
  );
  try {
    return await sesClient.send(sendReminderEmailCommand);
  } catch (caught) {
    if (caught instanceof Error && caught.name === "MessageRejected") {
      /** @type { import('@aws-sdk/client-ses').MessageRejected } */
      const messageRejectedError = caught;
      return messageRejectedError;
    }
    throw caught;
  }
};
```

- For API details, see [SendTemplatedEmail](#) in *AWS SDK for JavaScript API Reference*.

UpdateTemplate

The following code example shows how to use UpdateTemplate.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { UpdateTemplateCommand } from "@aws-sdk/client-ses";
import { getUniqueName } from "@aws-doc-sdk-examples/lib/utils/util-string.js";
import { sesClient } from "../libs/sesClient.js";

const TEMPLATE_NAME = getUniqueName("TemplateName");
const HTML_PART = "<h1>Hello, World!</h1>";

const createUpdateTemplateCommand = () => {
  return new UpdateTemplateCommand({
    Template: {
      TemplateName: TEMPLATE_NAME,
      HtmlPart: HTML_PART,
      SubjectPart: "Example",
      TextPart: "Updated template text.",
    },
  });
};

const run = async () => {
  const updateTemplateCommand = createUpdateTemplateCommand();

  try {
    return await sesClient.send(updateTemplateCommand);
  } catch (err) {
    console.log("Failed to update template.", err);
    return err;
  }
};
```

- For API details, see [UpdateTemplate](#) in *AWS SDK for JavaScript API Reference*.

VerifyDomainIdentity

The following code example shows how to use `VerifyDomainIdentity`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { VerifyDomainIdentityCommand } from "@aws-sdk/client-ses";
import {
  getUniqueName,
  postfix,
} from "@aws-doc-sdk-examples/lib/utils/util-string.js";
import { sesClient } from "../libs/sesClient.js";

/**
 * You must have access to the domain's DNS settings to complete the
 * domain verification process.
 */
const DOMAIN_NAME = postfix(getUniqueName("Domain"), ".example.com");

const createVerifyDomainIdentityCommand = () => {
  return new VerifyDomainIdentityCommand({ Domain: DOMAIN_NAME });
};

const run = async () => {
  const VerifyDomainIdentityCommand = createVerifyDomainIdentityCommand();

  try {
    return await sesClient.send(VerifyDomainIdentityCommand);
  } catch (err) {
    console.log("Failed to verify domain.", err);
    return err;
  }
};
```

- For API details, see [VerifyDomainIdentity](#) in *AWS SDK for JavaScript API Reference*.

VerifyEmailIdentity

The following code example shows how to use `VerifyEmailIdentity`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// Import required AWS SDK clients and commands for Node.js
import { VerifyEmailIdentityCommand } from "@aws-sdk/client-ses";
import { sesClient } from "../libs/sesClient.js";

const EMAIL_ADDRESS = "name@example.com";

const createVerifyEmailIdentityCommand = (emailAddress) => {
  return new VerifyEmailIdentityCommand({ EmailAddress: emailAddress });
};

const run = async () => {
  const verifyEmailIdentityCommand =
    createVerifyEmailIdentityCommand(EMAIL_ADDRESS);
  try {
    return await sesClient.send(verifyEmailIdentityCommand);
  } catch (err) {
    console.log("Failed to verify email identity.", err);
    return err;
  }
};
```

- For API details, see [VerifyEmailIdentity](#) in *AWS SDK for JavaScript API Reference*.

Scenarios

Build an Amazon Transcribe streaming app

The following code example shows how to build an app that records, transcribes, and translates live audio in real-time, and emails the results.

SDK for JavaScript (v3)

Shows how to use Amazon Transcribe to build an app that records, transcribes, and translates live audio in real-time, and emails the results using Amazon Simple Email Service (Amazon SES).

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- Amazon Comprehend
- Amazon SES
- Amazon Transcribe
- Amazon Translate

Create a web application to track DynamoDB data

The following code example shows how to create a web application that tracks work items in an Amazon DynamoDB table and uses Amazon Simple Email Service (Amazon SES) to send reports.

SDK for JavaScript (v3)

Shows how to use the Amazon DynamoDB API to create a dynamic web application that tracks DynamoDB work data.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- DynamoDB
- Amazon SES

Create an Aurora Serverless work item tracker

The following code example shows how to create a web application that tracks work items in an Amazon Aurora Serverless database and uses Amazon Simple Email Service (Amazon SES) to send reports.

SDK for JavaScript (v3)

Shows how to use the AWS SDK for JavaScript (v3) to create a web application that tracks work items in an Amazon Aurora database and emails reports by using Amazon Simple Email Service (Amazon SES). This example uses a front end built with React.js to interact with an Express Node.js backend.

- Integrate a React.js web application with AWS services.
- List, add, and update items in an Aurora table.
- Send an email report of filtered work items by using Amazon SES.
- Deploy and manage example resources with the included AWS CloudFormation script.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- Aurora
- Amazon RDS
- Amazon RDS Data Service
- Amazon SES

Detect PPE in images

The following code example shows how to build an app that uses Amazon Rekognition to detect Personal Protective Equipment (PPE) in images.

SDK for JavaScript (v3)

Shows how to use Amazon Rekognition with the AWS SDK for JavaScript to create an application to detect personal protective equipment (PPE) in images located in an Amazon Simple Storage Service (Amazon S3) bucket. The app saves the results to an Amazon

DynamoDB table, and sends the admin an email notification with the results using Amazon Simple Email Service (Amazon SES).

Learn how to:

- Create an unauthenticated user using Amazon Cognito.
- Analyze images for PPE using Amazon Rekognition.
- Verify an email address for Amazon SES.
- Update a DynamoDB table with results.
- Send an email notification using Amazon SES.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- DynamoDB
- Amazon Rekognition
- Amazon S3
- Amazon SES

Detect objects in images

The following code example shows how to build an app that uses Amazon Rekognition to detect objects by category in images.

SDK for JavaScript (v3)

Shows how to use Amazon Rekognition with the AWS SDK for JavaScript to create an app that uses Amazon Rekognition to identify objects by category in images located in an Amazon Simple Storage Service (Amazon S3) bucket. The app sends the admin an email notification with the results using Amazon Simple Email Service (Amazon SES).

Learn how to:

- Create an unauthenticated user using Amazon Cognito.
- Analyze images for objects using Amazon Rekognition.
- Verify an email address for Amazon SES.
- Send an email notification using Amazon SES.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- Amazon Rekognition
- Amazon S3
- Amazon SES

Amazon SNS examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with Amazon SNS.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Scenarios are code examples that show you how to accomplish specific tasks by calling multiple functions within a service or combined with other AWS services.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Get started

Hello Amazon SNS

The following code examples show how to get started using Amazon SNS.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Initialize an SNS client and and list topics in your account.

```
import { SNSClient, paginateListTopics } from "@aws-sdk/client-sns";
```

```
export const helloSns = async () => {
  // The configuration object ( `{}` ) is required. If the region and credentials
  // are omitted, the SDK uses your local configuration if it exists.
  const client = new SNSClient({});

  // You can also use `ListTopicsCommand`, but to use that command you must
  // handle the pagination yourself. You can do that by sending the
  `ListTopicsCommand`
  // with the `NextToken` parameter from the previous request.
  const paginatedTopics = paginateListTopics({ client }, {});
  const topics = [];

  for await (const page of paginatedTopics) {
    if (page.Topics?.length) {
      topics.push(...page.Topics);
    }
  }

  const suffix = topics.length === 1 ? "" : "s";

  console.log(
    `Hello, Amazon SNS! You have ${topics.length} topic${suffix} in your account.`
  );
  console.log(topics.map((t) => ` * ${t.TopicArn}`).join("\n"));
};
```

- For API details, see [ListTopics](#) in *AWS SDK for JavaScript API Reference*.

Topics

- [Actions](#)
- [Scenarios](#)
- [Serverless examples](#)

Actions

CheckIfPhoneNumberIsOptedOut

The following code example shows how to use `CheckIfPhoneNumberIsOptedOut`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create the client in a separate module and export it.

```
import { SNSClient } from "@aws-sdk/client-sns";

// The AWS Region can be provided here using the `region` property. If you leave it
// blank
// the SDK will default to the region set in your AWS config.
export const snsClient = new SNSClient({});
```

Import the SDK and client modules and call the API.

```
import { CheckIfPhoneNumberIsOptedOutCommand } from "@aws-sdk/client-sns";

import { snsClient } from "../libs/snsClient.js";

export const checkIfPhoneNumberIsOptedOut = async (
  phoneNumber = "5555555555",
) => {
  const command = new CheckIfPhoneNumberIsOptedOutCommand({
    phoneNumber,
  });

  const response = await snsClient.send(command);
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: '3341c28a-cdc8-5b39-a3ee-9fb0ee125732',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
```

```
// isOptedOut: false
// }
return response;
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [CheckIfPhoneNumberIsOptedOut](#) in *AWS SDK for JavaScript API Reference*.

ConfirmSubscription

The following code example shows how to use ConfirmSubscription.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create the client in a separate module and export it.

```
import { SNSClient } from "@aws-sdk/client-sns";

// The AWS Region can be provided here using the `region` property. If you leave it
// blank
// the SDK will default to the region set in your AWS config.
export const snsClient = new SNSClient({});
```

Import the SDK and client modules and call the API.

```
import { ConfirmSubscriptionCommand } from "@aws-sdk/client-sns";
import { snsClient } from "../libs/snsClient.js";

/**
 * @param {string} token - This token is sent the subscriber. Only subscribers
 * that are not AWS services (HTTP/S, email) need to be
 * confirmed.
```

```
* @param {string} topicArn - The ARN of the topic for which you wish to confirm a
subscription.
*/
export const confirmSubscription = async (
  token = "TOKEN",
  topicArn = "TOPIC_ARN",
) => {
  const response = await snsClient.send(
    // A subscription only needs to be confirmed if the endpoint type is
    // HTTP/S, email, or in another AWS account.
    new ConfirmSubscriptionCommand({
      Token: token,
      TopicArn: topicArn,
      // If this is true, the subscriber cannot unsubscribe while unauthenticated.
      AuthenticateOnUnsubscribe: "false",
    }),
  );
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: '4bb5bce9-805a-5517-8333-e1d2cface90b',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  //   SubscriptionArn: 'arn:aws:sns:us-east-1:xxxxxxxxxxxx:TOPIC_NAME:xxxxxxxx-
  xxx-xxxx-xxxx-xxxxxxxxxxxxx'
  // }
  return response;
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [ConfirmSubscription](#) in *AWS SDK for JavaScript API Reference*.

CreateTopic

The following code example shows how to use CreateTopic.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create the client in a separate module and export it.

```
import { SNSClient } from "@aws-sdk/client-sns";

// The AWS Region can be provided here using the `region` property. If you leave it
// blank
// the SDK will default to the region set in your AWS config.
export const snsClient = new SNSClient({});
```

Import the SDK and client modules and call the API.

```
import { CreateTopicCommand } from "@aws-sdk/client-sns";
import { snsClient } from "../libs/snsClient.js";

/**
 * @param {string} topicName - The name of the topic to create.
 */
export const createTopic = async (topicName = "TOPIC_NAME") => {
  const response = await snsClient.send(
    new CreateTopicCommand({ Name: topicName }),
  );
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: '087b8ad2-4593-50c4-a496-d7e90b82cf3e',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  //   TopicArn: 'arn:aws:sns:us-east-1:xxxxxxxxxxxx:TOPIC_NAME'
  // }
}
```

```
    return response;
  };
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [CreateTopic](#) in *AWS SDK for JavaScript API Reference*.

DeleteTopic

The following code example shows how to use DeleteTopic.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create the client in a separate module and export it.

```
import { SNSClient } from "@aws-sdk/client-sns";

// The AWS Region can be provided here using the `region` property. If you leave it
// blank
// the SDK will default to the region set in your AWS config.
export const snsClient = new SNSClient({});
```

Import the SDK and client modules and call the API.

```
import { DeleteTopicCommand } from "@aws-sdk/client-sns";
import { snsClient } from "../libs/snsClient.js";

/**
 * @param {string} topicArn - The ARN of the topic to delete.
 */
export const deleteTopic = async (topicArn = "TOPIC_ARN") => {
  const response = await snsClient.send(
    new DeleteTopicCommand({ TopicArn: topicArn }),
```

```
);
console.log(response);
// {
//   '$metadata': {
//     httpStatusCode: 200,
//     requestId: 'a10e2886-5a8f-5114-af36-75bd39498332',
//     extendedRequestId: undefined,
//     cfId: undefined,
//     attempts: 1,
//     totalRetryDelay: 0
//   }
// }
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [DeleteTopic](#) in *AWS SDK for JavaScript API Reference*.

GetSMSAttributes

The following code example shows how to use `GetSMSAttributes`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create the client in a separate module and export it.

```
import { SNSClient } from "@aws-sdk/client-sns";

// The AWS Region can be provided here using the `region` property. If you leave it
// blank
// the SDK will default to the region set in your AWS config.
export const snsClient = new SNSClient({});
```

Import the SDK and client modules and call the API.

```
import { GetSMSAttributesCommand } from "@aws-sdk/client-sns";
import { snsClient } from "../libs/snsClient.js";

export const getSmsAttributes = async () => {
  const response = await snsClient.send(
    // If you have not modified the account-level mobile settings of SNS,
    // the DefaultSMSType is undefined. For this example, it was set to
    // Transactional.
    new GetSMSAttributesCommand({ attributes: ["DefaultSMSType"] }),
  );

  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: '67ad8386-4169-58f1-bdb9-debd281d48d5',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  //   attributes: { DefaultSMSType: 'Transactional' }
  // }
  return response;
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [GetSMSAttributes](#) in *AWS SDK for JavaScript API Reference*.

GetTopicAttributes

The following code example shows how to use `GetTopicAttributes`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create the client in a separate module and export it.

```
import { SNSClient } from "@aws-sdk/client-sns";

// The AWS Region can be provided here using the `region` property. If you leave it
// blank
// the SDK will default to the region set in your AWS config.
export const snsClient = new SNSClient({});
```

Import the SDK and client modules and call the API.

```
import { GetTopicAttributesCommand } from "@aws-sdk/client-sns";
import { snsClient } from "../libs/snsClient.js";

/**
 * @param {string} topicArn - The ARN of the topic to retrieve attributes for.
 */
export const getTopicAttributes = async (topicArn = "TOPIC_ARN") => {
  const response = await snsClient.send(
    new GetTopicAttributesCommand({
      TopicArn: topicArn,
    }),
  );
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: '36b6a24e-5473-5d4e-ac32-ff72d9a73d94',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  //   Attributes: {
  //     Policy: '{...}',
  //     Owner: 'xxxxxxxxxxxxx',
  //     SubscriptionsPending: '1',
  //     TopicArn: 'arn:aws:sns:us-east-1:xxxxxxxxxxxxx:mytopic',
  //     TracingConfig: 'PassThrough',
  //     EffectiveDeliveryPolicy: '{"http":{"defaultHealthyRetryPolicy":
{"minDelayTarget":20,"maxDelayTarget":20,"numRetries":3,"numMaxDelayRetries":0,"numNoDelayRe
{"headerContentType":"text/plain; charset=UTF-8"}}}',
```



```
// SubscriptionsConfirmed: '0',  
// DisplayName: '',  
// SubscriptionsDeleted: '1'  
// }  
// }  
return response;  
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [GetTopicAttributes](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Import the SDK and client modules and call the API.

```
// 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);  
  });
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [GetTopicAttributes](#) in *AWS SDK for JavaScript API Reference*.

ListSubscriptions

The following code example shows how to use ListSubscriptions.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create the client in a separate module and export it.

```
import { SNSClient } from "@aws-sdk/client-sns";

// The AWS Region can be provided here using the `region` property. If you leave it
// blank
// the SDK will default to the region set in your AWS config.
export const snsClient = new SNSClient({});
```

Import the SDK and client modules and call the API.

```
import { ListSubscriptionsByTopicCommand } from "@aws-sdk/client-sns";
import { snsClient } from "../libs/snsClient.js";

/**
 * @param {string} topicArn - The ARN of the topic for which you wish to list
 * subscriptions.
 */
export const listSubscriptionsByTopic = async (topicArn = "TOPIC_ARN") => {
  const response = await snsClient.send(
    new ListSubscriptionsByTopicCommand({ TopicArn: topicArn }),
  );

  console.log(response);
  // {
```

```
// '$metadata': {
//   httpStatusCode: 200,
//   requestId: '0934fedf-0c4b-572e-9ed2-a3e38fadb0c8',
//   extendedRequestId: undefined,
//   cfId: undefined,
//   attempts: 1,
//   totalRetryDelay: 0
// },
// Subscriptions: [
//   {
//     SubscriptionArn: 'PendingConfirmation',
//     Owner: '901487484989',
//     Protocol: 'email',
//     Endpoint: 'corepyle@amazon.com',
//     TopicArn: 'arn:aws:sns:us-east-1:901487484989:mytopic'
//   }
// ]
// }
return response;
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [ListSubscriptions](#) in *AWS SDK for JavaScript API Reference*.

ListTopics

The following code example shows how to use ListTopics.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create the client in a separate module and export it.

```
import { SNSClient } from "@aws-sdk/client-sns";
```

```
// The AWS Region can be provided here using the `region` property. If you leave it
blank
// the SDK will default to the region set in your AWS config.
export const snsClient = new SNSClient({});
```

Import the SDK and client modules and call the API.

```
import { ListTopicsCommand } from "@aws-sdk/client-sns";
import { snsClient } from "../libs/snsClient.js";

export const listTopics = async () => {
  const response = await snsClient.send(new ListTopicsCommand({}));
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: '936bc5ad-83ca-53c2-b0b7-9891167b909e',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  //   Topics: [ { TopicArn: 'arn:aws:sns:us-east-1:xxxxxxxxxxxx:mytopic' } ]
  // }
  return response;
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [ListTopics](#) in *AWS SDK for JavaScript API Reference*.

Publish

The following code example shows how to use Publish.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create the client in a separate module and export it.

```
import { SNSClient } from "@aws-sdk/client-sns";

// The AWS Region can be provided here using the `region` property. If you leave it
// blank
// the SDK will default to the region set in your AWS config.
export const snsClient = new SNSClient({});
```

Import the SDK and client modules and call the API.

```
import { PublishCommand } from "@aws-sdk/client-sns";
import { snsClient } from "../libs/snsClient.js";

/**
 * @param {string | Record<string, any>} message - The message to send. Can be a
plain string or an object
 *
 *                                     if you are using the `json`
 * `MessageStructure`.
 * @param {string} topicArn - The ARN of the topic to which you would like to
publish.
 */
export const publish = async (
  message = "Hello from SNS!",
  topicArn = "TOPIC_ARN",
) => {
  const response = await snsClient.send(
    new PublishCommand({
      Message: message,
      TopicArn: topicArn,
    }),
  );
  console.log(response);
}
```

```
// {
//   '$metadata': {
//     httpStatusCode: 200,
//     requestId: 'e7f77526-e295-5325-9ee4-281a43ad1f05',
//     extendedRequestId: undefined,
//     cfId: undefined,
//     attempts: 1,
//     totalRetryDelay: 0
//   },
//   MessageId: 'xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxxx'
// }
return response;
};
```

Publish a message to a topic with group, duplication, and attribute options.

```
async publishMessages() {
  const message = await this.prompter.input({
    message: MESSAGES.publishMessagePrompt,
  });

  let groupId, deduplicationId, choices;

  if (this.isFifo) {
    await this.logger.log(MESSAGES.groupIdNotice);
    groupId = await this.prompter.input({
      message: MESSAGES.groupIdPrompt,
    });
  }

  if (this.autoDedup === false) {
    await this.logger.log(MESSAGES.deduplicationIdNotice);
    deduplicationId = await this.prompter.input({
      message: MESSAGES.deduplicationIdPrompt,
    });
  }

  choices = await this.prompter.checkbox({
    message: MESSAGES.messageAttributesPrompt,
    choices: toneChoices,
  });
}
```

```
await this.snsClient.send(
  new PublishCommand({
    TopicArn: this.topicArn,
    Message: message,
    ...(groupId
      ? {
          MessageGroupId: groupId,
        }
      : {}),
    ...(deduplicationId
      ? {
          MessageDeduplicationId: deduplicationId,
        }
      : {}),
    ...(choices
      ? {
          MessageAttributes: {
            tone: {
              DataType: "String.Array",
              StringValue: JSON.stringify(choices),
            },
          },
        }
      : {}),
  })),
);

const publishAnother = await this.prompter.confirm({
  message: MESSAGES.publishAnother,
});

if (publishAnother) {
  await this.publishMessages();
}
}
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [Publish](#) in *AWS SDK for JavaScript API Reference*.

SetSMSAttributes

The following code example shows how to use SetSMSAttributes.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create the client in a separate module and export it.

```
import { SNSClient } from "@aws-sdk/client-sns";

// The AWS Region can be provided here using the `region` property. If you leave it
// blank
// the SDK will default to the region set in your AWS config.
export const snsClient = new SNSClient({});
```

Import the SDK and client modules and call the API.

```
import { SetSMSAttributesCommand } from "@aws-sdk/client-sns";
import { snsClient } from "../libs/snsClient.js";

/**
 * @param {"Transactional" | "Promotional"} defaultSmsType
 */
export const setSmsType = async (defaultSmsType = "Transactional") => {
  const response = await snsClient.send(
    new SetSMSAttributesCommand({
      attributes: {
        // Promotional - (Default) Noncritical messages, such as marketing messages.
        // Transactional - Critical messages that support customer transactions,
        // such as one-time passcodes for multi-factor authentication.
        DefaultSMSType: defaultSmsType,
      },
    }),
  );
  console.log(response);
};
```



```
// {
//   '$metadata': {
//     httpStatusCode: 200,
//     requestId: '1885b977-2d7e-535e-8214-e44be727e265',
//     extendedRequestId: undefined,
//     cfId: undefined,
//     attempts: 1,
//     totalRetryDelay: 0
//   }
// }
return response;
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [SetSMSAttributes](#) in *AWS SDK for JavaScript API Reference*.

SetTopicAttributes

The following code example shows how to use SetTopicAttributes.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create the client in a separate module and export it.

```
import { SNSClient } from "@aws-sdk/client-sns";

// The AWS Region can be provided here using the `region` property. If you leave it
// blank
// the SDK will default to the region set in your AWS config.
export const snsClient = new SNSClient({});
```

Import the SDK and client modules and call the API.

```
import { SetTopicAttributesCommand } from "@aws-sdk/client-sns";
import { snsClient } from "../libs/snsClient.js";

export const setTopicAttributes = async (
  topicArn = "TOPIC_ARN",
  attributeName = "DisplayName",
  attributeValue = "Test Topic",
) => {
  const response = await snsClient.send(
    new SetTopicAttributesCommand({
      AttributeName: attributeName,
      AttributeValue: attributeValue,
      TopicArn: topicArn,
    }),
  );
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: 'd1b08d0e-e9a4-54c3-b8b1-d03238d2b935',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   }
  // }
  return response;
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [SetTopicAttributes](#) in *AWS SDK for JavaScript API Reference*.

Subscribe

The following code example shows how to use Subscribe.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create the client in a separate module and export it.

```
import { SNSClient } from "@aws-sdk/client-sns";

// The AWS Region can be provided here using the `region` property. If you leave it
// blank
// the SDK will default to the region set in your AWS config.
export const snsClient = new SNSClient({});
```

Import the SDK and client modules and call the API.

```
import { SubscribeCommand } from "@aws-sdk/client-sns";
import { snsClient } from "../libs/snsClient.js";

/**
 * @param {string} topicArn - The ARN of the topic for which you wish to confirm a
 * subscription.
 * @param {string} emailAddress - The email address that is subscribed to the topic.
 */
export const subscribeEmail = async (
  topicArn = "TOPIC_ARN",
  emailAddress = "usern@me.com",
) => {
  const response = await snsClient.send(
    new SubscribeCommand({
      Protocol: "email",
      TopicArn: topicArn,
      Endpoint: emailAddress,
    }),
  );
  console.log(response);
  // {
```

```
// '$metadata': {  
//   httpStatusCode: 200,  
//   requestId: 'c8e35bcd-b3c0-5940-9f66-06f6fcc108f0',  
//   extendedRequestId: undefined,  
//   cfId: undefined,  
//   attempts: 1,  
//   totalRetryDelay: 0  
// },  
// SubscriptionArn: 'pending confirmation'  
// }  
};
```

Subscribe a mobile application to a topic.

```
import { SubscribeCommand } from "@aws-sdk/client-sns";  
import { snsClient } from "../libs/snsClient.js";  
  
/**  
 * @param {string} topicArn - The ARN of the topic the subscriber is subscribing to.  
 * @param {string} endpoint - The Endpoint ARN of an application. This endpoint is  
 * created  
 * when an application registers for notifications.  
 */  
export const subscribeApp = async (  
  topicArn = "TOPIC_ARN",  
  endpoint = "ENDPOINT",  
) => {  
  const response = await snsClient.send(  
    new SubscribeCommand({  
      Protocol: "application",  
      TopicArn: topicArn,  
      Endpoint: endpoint,  
    })),  
  );  
  console.log(response);  
  // {  
  //   '$metadata': {  
  //     httpStatusCode: 200,  
  //     requestId: 'c8e35bcd-b3c0-5940-9f66-06f6fcc108f0',  
  //     extendedRequestId: undefined,  
  //     cfId: undefined,  
  //     attempts: 1,  
  //   },  
  //   SubscriptionArn: 'pending confirmation'  
  // }  
};
```

```
//     totalRetryDelay: 0
//   },
//   SubscriptionArn: 'pending confirmation'
// }
return response;
};
```

Subscribe a Lambda function to a topic.

```
import { SubscribeCommand } from "@aws-sdk/client-sns";
import { snsClient } from "../libs/snsClient.js";

/**
 * @param {string} topicArn - The ARN of the topic the subscriber is subscribing to.
 * @param {string} endpoint - The Endpoint ARN of an AWS Lambda function.
 */
export const subscribeLambda = async (
  topicArn = "TOPIC_ARN",
  endpoint = "ENDPOINT",
) => {
  const response = await snsClient.send(
    new SubscribeCommand({
      Protocol: "lambda",
      TopicArn: topicArn,
      Endpoint: endpoint,
    }),
  );
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: 'c8e35bcd-b3c0-5940-9f66-06f6fcc108f0',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  //   SubscriptionArn: 'pending confirmation'
  // }
  return response;
};
```

Subscribe an SQS queue to a topic.

```
import { SubscribeCommand, SNSClient } from "@aws-sdk/client-sns";

const client = new SNSClient({});

export const subscribeQueue = async (
  topicArn = "TOPIC_ARN",
  queueArn = "QUEUE_ARN",
) => {
  const command = new SubscribeCommand({
    TopicArn: topicArn,
    Protocol: "sqs",
    Endpoint: queueArn,
  });

  const response = await client.send(command);
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: '931e13d9-5e2b-543f-8781-4e9e494c5ff2',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  //   SubscriptionArn: 'arn:aws:sns:us-east-1:xxxxxxxxxxxx:subscribe-queue-
  test-430895:xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxxx'
  // }
  return response;
};
```

Subscribe with a filter to a topic.

```
import { SubscribeCommand, SNSClient } from "@aws-sdk/client-sns";

const client = new SNSClient({});

export const subscribeQueueFiltered = async (
  topicArn = "TOPIC_ARN",
  queueArn = "QUEUE_ARN",
```

```
) => {
  const command = new SubscribeCommand({
    TopicArn: topicArn,
    Protocol: "sqs",
    Endpoint: queueArn,
    Attributes: {
      // This subscription will only receive messages with the 'event' attribute set
      // to 'order_placed'.
      FilterPolicyScope: "MessageAttributes",
      FilterPolicy: JSON.stringify({
        event: ["order_placed"],
      }),
    },
  });

  const response = await client.send(command);
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: '931e13d9-5e2b-543f-8781-4e9e494c5ff2',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  //   SubscriptionArn: 'arn:aws:sns:us-east-1:xxxxxxxxxxxx:subscribe-queue-
  // test-430895:xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx'
  // }
  return response;
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [Subscribe](#) in *AWS SDK for JavaScript API Reference*.

Unsubscribe

The following code example shows how to use Unsubscribe.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create the client in a separate module and export it.

```
import { SNSClient } from "@aws-sdk/client-sns";

// The AWS Region can be provided here using the `region` property. If you leave it
// blank
// the SDK will default to the region set in your AWS config.
export const snsClient = new SNSClient({});
```

Import the SDK and client modules and call the API.

```
import { UnsubscribeCommand } from "@aws-sdk/client-sns";
import { snsClient } from "../libs/snsClient.js";

/**
 * @param {string} subscriptionArn - The ARN of the subscription to cancel.
 */
const unsubscribe = async (
  subscriptionArn = "arn:aws:sns:us-east-1:xxxxxxxxxxxx:mytopic:xxxxxxxx-xxxx-xxxx-
  xxxx-xxxxxxxxxxxx",
) => {
  const response = await snsClient.send(
    new UnsubscribeCommand({
      SubscriptionArn: subscriptionArn,
    }),
  );
  console.log(response);
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: '0178259a-9204-507c-b620-78a7570a44c6',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
```



```
//     attempts: 1,  
//     totalRetryDelay: 0  
//   }  
// }  
return response;  
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [Unsubscribe](#) in *AWS SDK for JavaScript API Reference*.

Scenarios

Build an app to submit data to a DynamoDB table

The following code example shows how to build an application that submits data to an Amazon DynamoDB table and notifies you when a user updates the table.

SDK for JavaScript (v3)

This example shows how to build an app that enables users to submit data to an Amazon DynamoDB table, and send a text message to the administrator using Amazon Simple Notification Service (Amazon SNS).

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

This example is also available in the [AWS SDK for JavaScript v3 developer guide](#).

Services used in this example

- DynamoDB
- Amazon SNS

Create a serverless application to manage photos

The following code example shows how to create a serverless application that lets users manage photos using labels.

SDK for JavaScript (v3)

Shows how to develop a photo asset management application that detects labels in images using Amazon Rekognition and stores them for later retrieval.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

For a deep dive into the origin of this example see the post on [AWS Community](#).

Services used in this example

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Create an Amazon Textract explorer application

The following code example shows how to explore Amazon Textract output through an interactive application.

SDK for JavaScript (v3)

Shows how to use the AWS SDK for JavaScript to build a React application that uses Amazon Textract to extract data from a document image and display it in an interactive web page. This example runs in a web browser and requires an authenticated Amazon Cognito identity for credentials. It uses Amazon Simple Storage Service (Amazon S3) for storage, and for notifications it polls an Amazon Simple Queue Service (Amazon SQS) queue that is subscribed to an Amazon Simple Notification Service (Amazon SNS) topic.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- Amazon Cognito Identity
- Amazon S3

- Amazon SNS
- Amazon SQS
- Amazon Textract

Publish messages to queues

The following code example shows how to:

- Create topic (FIFO or non-FIFO).
- Subscribe several queues to the topic with an option to apply a filter.
- Publish messages to the topic.
- Poll the queues for messages received.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

This is the entry point for this workflow.

```
import { SNSClient } from "@aws-sdk/client-sns";
import { SQSClient } from "@aws-sdk/client-sqs";

import { TopicsQueuesWkflw } from "./TopicsQueuesWkflw.js";
import { Prompter } from "@aws-doc-sdk-examples/lib/prompter.js";
import { SlowLogger } from "@aws-doc-sdk-examples/lib/slow-logger.js";

export const startSnsWorkflow = () => {
  const noLoggerDelay = process.argv.find((arg) => arg === "--no-logger-delay");
  const snsClient = new SNSClient({});
  const sqsClient = new SQSClient({});
  const prompter = new Prompter();
  const logger = noLoggerDelay ? console : new SlowLogger(25);

  const wkflw = new TopicsQueuesWkflw(snsClient, sqsClient, prompter, logger);
```

```
wkflw.start();  
};
```

The preceding code provides the necessary dependencies and starts the workflow. The next section contains the bulk of the example.

```
const toneChoices = [  
  { name: "cheerful", value: "cheerful" },  
  { name: "funny", value: "funny" },  
  { name: "serious", value: "serious" },  
  { name: "sincere", value: "sincere" },  
];  
  
export class TopicsQueuesWkflw {  
  // SNS topic is configured as First-In-First-Out  
  isFifo = true;  
  
  // Automatic content-based deduplication is enabled.  
  autoDedup = false;  
  
  snsClient;  
  sqsClient;  
  topicName;  
  topicArn;  
  subscriptionArns = [];  
  /**  
   * @type {{ queueName: string, queueArn: string, queueUrl: string, policy?:  
string }[]}  
   */  
  queues = [];  
  prompter;  
  
  /**  
   * @param {import('@aws-sdk/client-sns').SNSClient} snsClient  
   * @param {import('@aws-sdk/client-sqs').SQSClient} sqsClient  
   * @param {import('../libs/prompter.js').Prompter} prompter  
   * @param {import('../libs/logger.js').Logger} logger  
   */  
  constructor(snsClient, sqsClient, prompter, logger) {  
    this.snsClient = snsClient;
```

```
    this.sqsClient = sqsClient;
    this.prompter = prompter;
    this.logger = logger;
  }

  async welcome() {
    await this.logger.log(MESSAGES.description);
  }

  async confirmFifo() {
    await this.logger.log(MESSAGES.snsFifoDescription);
    this.isFifo = await this.prompter.confirm({
      message: MESSAGES.snsFifoPrompt,
    });

    if (this.isFifo) {
      this.logger.logSeparator(MESSAGES.headerDedup);
      await this.logger.log(MESSAGES.deduplicationNotice);
      await this.logger.log(MESSAGES.deduplicationDescription);
      this.autoDedup = await this.prompter.confirm({
        message: MESSAGES.deduplicationPrompt,
      });
    }
  }

  async createTopic() {
    await this.logger.log(MESSAGES.creatingTopics);
    this.topicName = await this.prompter.input({
      message: MESSAGES.topicNamePrompt,
    });
    if (this.isFifo) {
      this.topicName += ".fifo";
      this.logger.logSeparator(MESSAGES.headerFifoNaming);
      await this.logger.log(MESSAGES.appendFifoNotice);
    }

    const response = await this.snsClient.send(
      new CreateTopicCommand({
        Name: this.topicName,
        Attributes: {
          FifoTopic: this.isFifo ? "true" : "false",
          ...(this.autoDedup ? { ContentBasedDeduplication: "true" } : {}),
        },
      }),
    ),
```

```
);

this.topicArn = response.TopicArn;

await this.logger.log(
  MESSAGES.topicCreatedNotice
    .replace("${TOPIC_NAME}", this.topicName)
    .replace("${TOPIC_ARN}", this.topicArn),
);
}

async createQueues() {
  await this.logger.log(MESSAGES.createQueuesNotice);
  // Increase this number to add more queues.
  let maxQueues = 2;

  for (let i = 0; i < maxQueues; i++) {
    await this.logger.log(MESSAGES.queueCount.replace("${COUNT}", i + 1));
    let queueName = await this.prompter.input({
      message: MESSAGES.queueNamePrompt.replace(
        "${EXAMPLE_NAME}",
        i === 0 ? "good-news" : "bad-news",
      ),
    });
  });

  if (this.isFifo) {
    queueName += ".fifo";
    await this.logger.log(MESSAGES.appendFifoNotice);
  }

  const response = await this.sqsClient.send(
    new CreateQueueCommand({
      QueueName: queueName,
      Attributes: { ...(this.isFifo ? { FifoQueue: "true" } : {}) },
    }),
  );

  const { Attributes } = await this.sqsClient.send(
    new GetQueueAttributesCommand({
      QueueUrl: response.QueueUrl,
      AttributeNames: ["QueueArn"],
    }),
  );
};
```

```
    this.queues.push({
      queueName,
      queueArn: Attributes.QueueArn,
      queueUrl: response.QueueUrl,
    });

    await this.logger.log(
      MESSAGES.queueCreatedNotice
        .replace("${QUEUE_NAME}", queueName)
        .replace("${QUEUE_URL}", response.QueueUrl)
        .replace("${QUEUE_ARN}", Attributes.QueueArn),
    );
  }
}

async attachQueueIamPolicies() {
  for (const [index, queue] of this.queues.entries()) {
    const policy = JSON.stringify(
      {
        Statement: [
          {
            Effect: "Allow",
            Principal: {
              Service: "sns.amazonaws.com",
            },
            Action: "sqs:SendMessage",
            Resource: queue.queueArn,
            Condition: {
              ArnEquals: {
                "aws:SourceArn": this.topicArn,
              },
            },
          },
        ],
      },
      null,
      2,
    );

    if (index !== 0) {
      this.logger.logSeparator();
    }

    await this.logger.log(MESSAGES.attachPolicyNotice);
  }
}
```

```
console.log(policy);
const addPolicy = await this.prompter.confirm({
  message: MESSAGES.addPolicyConfirmation.replace(
    "${QUEUE_NAME}",
    queue.queueName,
  ),
});

if (addPolicy) {
  await this.sqsClient.send(
    new SetQueueAttributesCommand({
      QueueUrl: queue.queueUrl,
      Attributes: {
        Policy: policy,
      },
    }),
  );
  queue.policy = policy;
} else {
  await this.logger.log(
    MESSAGES.policyNotAttachedNotice.replace(
      "${QUEUE_NAME}",
      queue.queueName,
    ),
  );
}
}

async subscribeQueuesToTopic() {
  for (const [index, queue] of this.queues.entries()) {
    /**
     * @type {import('@aws-sdk/client-sns').SubscribeCommandInput}
     */
    const subscribeParams = {
      TopicArn: this.topicArn,
      Protocol: "sqs",
      Endpoint: queue.queueArn,
    };
    let tones = [];

    if (this.isFifo) {
      if (index === 0) {
        await this.logger.log(MESSAGES.fifoFilterNotice);
      }
    }
  }
}
```



```
    }
    tones = await this.prompter.checkbox({
      message: MESSAGES.fifoFilterSelect.replace(
        "${QUEUE_NAME}",
        queue.queueName,
      ),
      choices: toneChoices,
    });

    if (tones.length) {
      subscribeParams.Attributes = {
        FilterPolicyScope: "MessageAttributes",
        FilterPolicy: JSON.stringify({
          tone: tones,
        }),
      };
    }
  }
}

const { SubscriptionArn } = await this.snsClient.send(
  new SubscribeCommand(subscribeParams),
);

this.subscriptionArns.push(SubscriptionArn);

await this.logger.log(
  MESSAGES.queueSubscribedNotice
    .replace("${QUEUE_NAME}", queue.queueName)
    .replace("${TOPIC_NAME}", this.topicName)
    .replace("${TONES}", tones.length ? tones.join(", ") : "none"),
);
}
}

async publishMessages() {
  const message = await this.prompter.input({
    message: MESSAGES.publishMessagePrompt,
  });

  let groupId, deduplicationId, choices;

  if (this.isFifo) {
    await this.logger.log(MESSAGES.groupIdNotice);
    groupId = await this.prompter.input({
```

```
    message: MESSAGES.groupIdPrompt,
  });

  if (this.autoDedup === false) {
    await this.logger.log(MESSAGES.deduplicationIdNotice);
    deduplicationId = await this.prompter.input({
      message: MESSAGES.deduplicationIdPrompt,
    });
  }

  choices = await this.prompter.checkbox({
    message: MESSAGES.messageAttributesPrompt,
    choices: toneChoices,
  });
}

await this.snsClient.send(
  new PublishCommand({
    TopicArn: this.topicArn,
    Message: message,
    ...(groupId
      ? {
          MessageGroupId: groupId,
        }
      : {}),
    ...(deduplicationId
      ? {
          MessageDeduplicationId: deduplicationId,
        }
      : {}),
    ...(choices
      ? {
          MessageAttributes: {
            tone: {
              DataType: "String.Array",
              StringValue: JSON.stringify(choices),
            },
          },
        }
      : {}),
  })),
);

const publishAnother = await this.prompter.confirm({
```

```
    message: MESSAGES.publishAnother,
  });

  if (publishAnother) {
    await this.publishMessages();
  }
}

async receiveAndDeleteMessages() {
  for (const queue of this.queues) {
    const { Messages } = await this.sqsClient.send(
      new ReceiveMessageCommand({
        QueueUrl: queue.queueUrl,
      }),
    );

    if (Messages) {
      await this.logger.log(
        MESSAGES.messagesReceivedNotice.replace(
          "${QUEUE_NAME}",
          queue.queueName,
        ),
      );
      console.log(Messages);

      await this.sqsClient.send(
        new DeleteMessageBatchCommand({
          QueueUrl: queue.queueUrl,
          Entries: Messages.map((message) => ({
            Id: message.MessageId,
            ReceiptHandle: message.ReceiptHandle,
          })),
        }),
      );
    } else {
      await this.logger.log(
        MESSAGES.noMessagesReceivedNotice.replace(
          "${QUEUE_NAME}",
          queue.queueName,
        ),
      );
    }
  }
}
```

```
const deleteAndPoll = await this.prompter.confirm({
  message: MESSAGES.deleteAndPollConfirmation,
});

if (deleteAndPoll) {
  await this.receiveAndDeleteMessages();
}

}

async destroyResources() {
  for (const subscriptionArn of this.subscriptionArns) {
    await this.snsClient.send(
      new UnsubscribeCommand({ SubscriptionArn: subscriptionArn } ),
    );
  }

  for (const queue of this.queues) {
    await this.sqsClient.send(
      new DeleteQueueCommand({ QueueUrl: queue.queueUrl } ),
    );
  }

  if (this.topicArn) {
    await this.snsClient.send(
      new DeleteTopicCommand({ TopicArn: this.topicArn } ),
    );
  }
}

async start() {
  console.clear();

  try {
    this.logger.logSeparator(MESSAGES.headerWelcome);
    await this.welcome();
    this.logger.logSeparator(MESSAGES.headerFifo);
    await this.confirmFifo();
    this.logger.logSeparator(MESSAGES.headerCreateTopic);
    await this.createTopic();
    this.logger.logSeparator(MESSAGES.headerCreateQueues);
    await this.createQueues();
    this.logger.logSeparator(MESSAGES.headerAttachPolicy);
    await this.attachQueueIamPolicies();
    this.logger.logSeparator(MESSAGES.headerSubscribeQueues);
```

```
    await this.subscribeQueuesToTopic();
    this.logger.logSeparator(MESSAGES.headerPublishMessage);
    await this.publishMessages();
    this.logger.logSeparator(MESSAGES.headerReceiveMessages);
    await this.receiveAndDeleteMessages();
  } catch (err) {
    console.error(err);
  } finally {
    await this.destroyResources();
  }
}
}
```

- For API details, see the following topics in *AWS SDK for JavaScript API Reference*.
 - [CreateQueue](#)
 - [CreateTopic](#)
 - [DeleteMessageBatch](#)
 - [DeleteQueue](#)
 - [DeleteTopic](#)
 - [GetQueueAttributes](#)
 - [Publish](#)
 - [ReceiveMessage](#)
 - [SetQueueAttributes](#)
 - [Subscribe](#)
 - [Unsubscribe](#)

Serverless examples

Invoke a Lambda function from an Amazon SNS trigger

The following code example shows how to implement a Lambda function that receives an event triggered by receiving messages from an SNS topic. The function retrieves the messages from the event parameter and logs the content of each message.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [Serverless examples](#) repository.

Consuming an SNS event with Lambda using JavaScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    await processMessageAsync(record);
  }
  console.info("done");
};

async function processMessageAsync(record) {
  try {
    const message = JSON.stringify(record.Sns.Message);
    console.log(`Processed message ${message}`);
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

Consuming an SNS event with Lambda using TypeScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SNSEvent, Context, SNSHandler, SNSEventRecord } from "aws-lambda";

export const functionHandler: SNSHandler = async (
  event: SNSEvent,
  context: Context
): Promise<void> => {
  for (const record of event.Records) {
```

```
    await processMessageAsync(record);
  }
  console.info("done");
};

async function processMessageAsync(record: SNSEventRecord): Promise<any> {
  try {
    const message: string = JSON.stringify(record.Sns.Message);
    console.log(`Processed message ${message}`);
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

Amazon SQS examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with Amazon SQS.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Scenarios are code examples that show you how to accomplish specific tasks by calling multiple functions within a service or combined with other AWS services.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Get started

Hello Amazon SQS

The following code examples show how to get started using Amazon SQS.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Initialize an Amazon SQS client and list queues.

```
import { SQSClient, paginateListQueues } from "@aws-sdk/client-sqs";

export const helloSqs = async () => {
  // The configuration object ( `{}` ) is required. If the region and credentials
  // are omitted, the SDK uses your local configuration if it exists.
  const client = new SQSClient({});

  // You can also use `ListQueuesCommand`, but to use that command you must
  // handle the pagination yourself. You can do that by sending the
  `ListQueuesCommand`
  // with the `NextToken` parameter from the previous request.
  const paginatedQueues = paginateListQueues({ client }, {});
  const queues = [];

  for await (const page of paginatedQueues) {
    if (page.QueueUrls?.length) {
      queues.push(...page.QueueUrls);
    }
  }

  const suffix = queues.length === 1 ? "" : "s";

  console.log(
    `Hello, Amazon SQS! You have ${queues.length} queue${suffix} in your account.`
  );
  console.log(queues.map((t) => ` * ${t}`).join("\n"));
};
```

- For API details, see [ListQueues](#) in *AWS SDK for JavaScript API Reference*.

Topics

- [Actions](#)
- [Scenarios](#)
- [Serverless examples](#)

Actions

ChangeMessageVisibility

The following code example shows how to use `ChangeMessageVisibility`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Receive an Amazon SQS message and change its timeout visibility.

```
import {
  ReceiveMessageCommand,
  ChangeMessageVisibilityCommand,
  SQSClient,
} from "@aws-sdk/client-sqs";

const client = new SQSClient({});
const SQS_QUEUE_URL = "queue_url";

const receiveMessage = (queueUrl) =>
  client.send(
    new ReceiveMessageCommand({
      AttributeNames: ["SentTimestamp"],
      MaxNumberOfMessages: 1,
      MessageAttributeNames: ["All"],
      QueueUrl: queueUrl,
      WaitTimeSeconds: 1,
    }),
  );

export const main = async (queueUrl = SQS_QUEUE_URL) => {
```

```
const { Messages } = await receiveMessage(queueUrl);

const response = await client.send(
  new ChangeMessageVisibilityCommand({
    QueueUrl: queueUrl,
    ReceiptHandle: Messages[0].ReceiptHandle,
    VisibilityTimeout: 20,
  }),
);
console.log(response);
return response;
};
```

- For API details, see [ChangeMessageVisibility](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Receive an Amazon SQS message and change its timeout visibility.

```
// 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");
    }
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [ChangeMessageVisibility](#) in *AWS SDK for JavaScript API Reference*.

CreateQueue

The following code example shows how to use CreateQueue.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create an Amazon SQS standard queue.

```
import { CreateQueueCommand, SQSClient } from "@aws-sdk/client-sqs";

const client = new SQSClient({});
const SQS_QUEUE_NAME = "test-queue";

export const main = async (sqsQueueName = SQS_QUEUE_NAME) => {
  const command = new CreateQueueCommand({
    QueueName: sqsQueueName,
    Attributes: {
      DelaySeconds: "60",
      MessageRetentionPeriod: "86400",
    },
  });

  const response = await client.send(command);
  console.log(response);
  return response;
};
```

Create an Amazon SQS queue with long polling.

```
import { CreateQueueCommand, SQSClient } from "@aws-sdk/client-sqs";

const client = new SQSClient({});
const SQS_QUEUE_NAME = "queue_name";

export const main = async (queueName = SQS_QUEUE_NAME) => {
  const response = await client.send(
    new CreateQueueCommand({
      QueueName: queueName,
      Attributes: {
        // When the wait time for the ReceiveMessage API action is greater than 0,
        // long polling is in effect. The maximum long polling wait time is 20
        // seconds. Long polling helps reduce the cost of using Amazon SQS by,
        // eliminating the number of empty responses and false empty responses.
        // https://docs.aws.amazon.com/AWSSimpleQueueService/latest/
        SQSDeveloperGuide/sqs-short-and-long-polling.html
        ReceiveMessageWaitTimeSeconds: "20",
      },
    })
  );
  console.log(response);
};
```

```
    return response;
  };
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [CreateQueue](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create an Amazon SQS standard queue.

```
// 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);
  }
});
```

Create an Amazon SQS queue that waits for a message to arrive.

```
// 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);
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [CreateQueue](#) in *AWS SDK for JavaScript API Reference*.

DeleteMessage

The following code example shows how to use DeleteMessage.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Receive and delete Amazon SQS messages.

```
import {
  ReceiveMessageCommand,
  DeleteMessageCommand,
  SQSClient,
  DeleteMessageBatchCommand,
} from "@aws-sdk/client-sqs";

const client = new SQSClient({});
const SQS_QUEUE_URL = "queue_url";

const receiveMessage = (queueUrl) =>
  client.send(
    new ReceiveMessageCommand({
      AttributeNames: ["SentTimestamp"],
      MaxNumberOfMessages: 10,
      MessageAttributeNames: ["All"],
      QueueUrl: queueUrl,
      WaitTimeSeconds: 20,
      VisibilityTimeout: 20,
    }),
  );

export const main = async (queueUrl = SQS_QUEUE_URL) => {
  const { Messages } = await receiveMessage(queueUrl);

  if (!Messages) {
    return;
  }

  if (Messages.length === 1) {
    console.log(Messages[0].Body);
    await client.send(
      new DeleteMessageCommand({
        QueueUrl: queueUrl,
        ReceiptHandle: Messages[0].ReceiptHandle,
      }),
    );
  } else {
    await client.send(
      new DeleteMessageBatchCommand({
        QueueUrl: queueUrl,
        Entries: Messages.map((message) => ({
```

```
        Id: message.MessageId,
        ReceiptHandle: message.ReceiptHandle,
    })),
    }),
);
}
```

- For API details, see [DeleteMessage](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Receive and delete Amazon SQS messages.

```
// 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);
        }
      });
    }
  });
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [DeleteMessage](#) in *AWS SDK for JavaScript API Reference*.

DeleteMessageBatch

The following code example shows how to use DeleteMessageBatch.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import {
  ReceiveMessageCommand,
  DeleteMessageCommand,
  SQSClient,
  DeleteMessageBatchCommand,
} from "@aws-sdk/client-sqs";

const client = new SQSClient({});
const SQS_QUEUE_URL = "queue_url";
```

```
const receiveMessage = (queueUrl) =>
  client.send(
    new ReceiveMessageCommand({
      AttributeNames: ["SentTimestamp"],
      MaxNumberOfMessages: 10,
      MessageAttributeNames: ["All"],
      QueueUrl: queueUrl,
      WaitTimeSeconds: 20,
      VisibilityTimeout: 20,
    }),
  );

export const main = async (queueUrl = SQS_QUEUE_URL) => {
  const { Messages } = await receiveMessage(queueUrl);

  if (!Messages) {
    return;
  }

  if (Messages.length === 1) {
    console.log(Messages[0].Body);
    await client.send(
      new DeleteMessageCommand({
        QueueUrl: queueUrl,
        ReceiptHandle: Messages[0].ReceiptHandle,
      }),
    );
  } else {
    await client.send(
      new DeleteMessageBatchCommand({
        QueueUrl: queueUrl,
        Entries: Messages.map((message) => ({
          Id: message.MessageId,
          ReceiptHandle: message.ReceiptHandle,
        })),
      }),
    );
  }
};
```

- For API details, see [DeleteMessageBatch](#) in *AWS SDK for JavaScript API Reference*.

DeleteQueue

The following code example shows how to use DeleteQueue.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Delete an Amazon SQS queue.

```
import { DeleteQueueCommand, SQSClient } from "@aws-sdk/client-sqs";

const client = new SQSClient({});
const SQS_QUEUE_URL = "test-queue-url";

export const main = async (queueUrl = SQS_QUEUE_URL) => {
  const command = new DeleteQueueCommand({ QueueUrl: queueUrl });

  const response = await client.send(command);
  console.log(response);
  return response;
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [DeleteQueue](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Delete an Amazon SQS queue.

```
// 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);
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [DeleteQueue](#) in *AWS SDK for JavaScript API Reference*.

GetQueueAttributes

The following code example shows how to use `GetQueueAttributes`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { GetQueueAttributesCommand, SQSClient } from "@aws-sdk/client-sqs";

const client = new SQSClient({});
const SQS_QUEUE_URL = "queue-url";

export const getQueueAttributes = async (queueUrl = SQS_QUEUE_URL) => {
```

```
const command = new GetQueueAttributesCommand({
  QueueUrl: queueUrl,
  AttributeNames: ["DelaySeconds"],
});

const response = await client.send(command);
console.log(response);
// {
//   '$metadata': {
//     httpStatusCode: 200,
//     requestId: '747a1192-c334-5682-a508-4cd5e8dc4e79',
//     extendedRequestId: undefined,
//     cfId: undefined,
//     attempts: 1,
//     totalRetryDelay: 0
//   },
//   Attributes: { DelaySeconds: '1' }
// }
return response;
};
```

- For API details, see [GetQueueAttributes](#) in *AWS SDK for JavaScript API Reference*.

GetQueueUrl

The following code example shows how to use `GetQueueUrl`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Get the URL for an Amazon SQS queue.

```
import { GetQueueUrlCommand, SQSClient } from "@aws-sdk/client-sqs";

const client = new SQSClient({});
const SQS_QUEUE_NAME = "test-queue";
```

```
export const main = async (queueName = SQS_QUEUE_NAME) => {
  const command = new GetQueueUrlCommand({ QueueName: queueName });

  const response = await client.send(command);
  console.log(response);
  return response;
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [GetQueueUrl](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Get the URL for an Amazon SQS queue.

```
// 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);
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [GetQueueUrl](#) in *AWS SDK for JavaScript API Reference*.

ListQueues

The following code example shows how to use ListQueues.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

List your Amazon SQS queues.

```
import { paginateListQueues, SQSClient } from "@aws-sdk/client-sqs";

const client = new SQSClient({});

export const main = async () => {
  const paginatedListQueues = paginateListQueues({ client }, {});

  /** @type {string[]} */
  const urls = [];
  for await (const page of paginatedListQueues) {
    const nextUrls = page.QueueUrls?.filter((qurl) => !!qurl) || [];
    urls.push(...nextUrls);
    urls.forEach((url) => console.log(url));
  }

  return urls;
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [ListQueues](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

List your Amazon SQS queues.

```
// 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);
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [ListQueues](#) in *AWS SDK for JavaScript API Reference*.

ReceiveMessage

The following code example shows how to use `ReceiveMessage`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Receive a message from an Amazon SQS queue.

```
import {
  ReceiveMessageCommand,
  DeleteMessageCommand,
  SQSClient,
  DeleteMessageBatchCommand,
} from "@aws-sdk/client-sqs";

const client = new SQSClient({});
const SQS_QUEUE_URL = "queue_url";

const receiveMessage = (queueUrl) =>
  client.send(
    new ReceiveMessageCommand({
      AttributeNames: ["SentTimestamp"],
      MaxNumberOfMessages: 10,
      MessageAttributeNames: ["All"],
      QueueUrl: queueUrl,
      WaitTimeSeconds: 20,
      VisibilityTimeout: 20,
    }),
  );

export const main = async (queueUrl = SQS_QUEUE_URL) => {
  const { Messages } = await receiveMessage(queueUrl);

  if (!Messages) {
    return;
  }

  if (Messages.length === 1) {
    console.log(Messages[0].Body);
    await client.send(
      new DeleteMessageCommand({
```

```
        QueueUrl: queueUrl,
        ReceiptHandle: Messages[0].ReceiptHandle,
    })),
    );
} else {
    await client.send(
        new DeleteMessageBatchCommand({
            QueueUrl: queueUrl,
            Entries: Messages.map((message) => ({
                Id: message.MessageId,
                ReceiptHandle: message.ReceiptHandle,
            })),
        })),
    );
}
};
```

Receive a message from an Amazon SQS queue using long-poll support.

```
import { ReceiveMessageCommand, SQSClient } from "@aws-sdk/client-sqs";

const client = new SQSClient({});
const SQS_QUEUE_URL = "queue-url";

export const main = async (queueUrl = SQS_QUEUE_URL) => {
    const command = new ReceiveMessageCommand({
        AttributeNames: ["SentTimestamp"],
        MaxNumberOfMessages: 1,
        MessageAttributeNames: ["All"],
        QueueUrl: queueUrl,
        // The duration (in seconds) for which the call waits for a message
        // to arrive in the queue before returning. If a message is available,
        // the call returns sooner than WaitTimeSeconds. If no messages are
        // available and the wait time expires, the call returns successfully
        // with an empty list of messages.
        // https://docs.aws.amazon.com/AWSSimpleQueueService/latest/APIReference/
        API_ReceiveMessage.html#API_ReceiveMessage_RequestSyntax
        WaitTimeSeconds: 20,
    });

    const response = await client.send(command);
    console.log(response);
};
```

```
    return response;
};
```

- For API details, see [ReceiveMessage](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Receive a message from an Amazon SQS queue using long-poll support.

```
// 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);
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [ReceiveMessage](#) in *AWS SDK for JavaScript API Reference*.

SendMessage

The following code example shows how to use SendMessage.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Send a message to an Amazon SQS queue.

```
import { SendMessageCommand, SQSClient } from "@aws-sdk/client-sqs";

const client = new SQSClient({});
const SQS_QUEUE_URL = "queue_url";

export const main = async (sqsQueueUrl = SQS_QUEUE_URL) => {
  const command = new SendMessageCommand({
    QueueUrl: sqsQueueUrl,
    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.",
  });

  const response = await client.send(command);
  console.log(response);
  return response;
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [SendMessage](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Send a message to an Amazon SQS queue.

```
// 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);
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [SendMessage](#) in *AWS SDK for JavaScript API Reference*.

SetQueueAttributes

The following code example shows how to use `SetQueueAttributes`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { SetQueueAttributesCommand, SQSClient } from "@aws-sdk/client-sqs";

const client = new SQSClient({});
const SQS_QUEUE_URL = "queue-url";
```

```
export const main = async (queueUrl = SQS_QUEUE_URL) => {
  const command = new SetQueueAttributesCommand({
    QueueUrl: queueUrl,
    Attributes: {
      DelaySeconds: "1",
    },
  });

  const response = await client.send(command);
  console.log(response);
  return response;
};
```

Configure an Amazon SQS queue to use long polling.

```
import { SetQueueAttributesCommand, SQSClient } from "@aws-sdk/client-sqs";

const client = new SQSClient({});
const SQS_QUEUE_URL = "queue_url";

export const main = async (queueUrl = SQS_QUEUE_URL) => {
  const command = new SetQueueAttributesCommand({
    Attributes: {
      ReceiveMessageWaitTimeSeconds: "20",
    },
    QueueUrl: queueUrl,
  });

  const response = await client.send(command);
  console.log(response);
  return response;
};
```

Configure a dead-letter queue.

```
import { SetQueueAttributesCommand, SQSClient } from "@aws-sdk/client-sqs";

const client = new SQSClient({});
const SQS_QUEUE_URL = "queue_url";
const DEAD_LETTER_QUEUE_ARN = "dead_letter_queue_arn";
```

```
export const main = async (
  queueUrl = SQS_QUEUE_URL,
  deadLetterQueueArn = DEAD_LETTER_QUEUE_ARN,
) => {
  const command = new SetQueueAttributesCommand({
    Attributes: {
      RedrivePolicy: JSON.stringify({
        // Amazon SQS supports dead-letter queues (DLQ), which other
        // queues (source queues) can target for messages that can't
        // be processed (consumed) successfully.
        // https://docs.aws.amazon.com/AWSSimpleQueueService/latest/
        SQSDeveloperGuide/sqs-dead-letter-queues.html
        deadLetterTargetArn: deadLetterQueueArn,
        maxReceiveCount: "10",
      }),
    },
    QueueUrl: queueUrl,
  });

  const response = await client.send(command);
  console.log(response);
  return response;
};
```

- For API details, see [SetQueueAttributes](#) in *AWS SDK for JavaScript API Reference*.

Scenarios

Create an Amazon Textract explorer application

The following code example shows how to explore Amazon Textract output through an interactive application.

SDK for JavaScript (v3)

Shows how to use the AWS SDK for JavaScript to build a React application that uses Amazon Textract to extract data from a document image and display it in an interactive web page. This example runs in a web browser and requires an authenticated Amazon Cognito identity for credentials. It uses Amazon Simple Storage Service (Amazon S3) for storage, and for notifications it polls an Amazon Simple Queue Service (Amazon SQS) queue that is subscribed to an Amazon Simple Notification Service (Amazon SNS) topic.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- Amazon Cognito Identity
- Amazon S3
- Amazon SNS
- Amazon SQS
- Amazon Textract

Publish messages to queues

The following code example shows how to:

- Create topic (FIFO or non-FIFO).
- Subscribe several queues to the topic with an option to apply a filter.
- Publish messages to the topic.
- Poll the queues for messages received.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

This is the entry point for this workflow.

```
import { SNSClient } from "@aws-sdk/client-sns";
import { SQSClient } from "@aws-sdk/client-sqs";

import { TopicsQueuesWkflw } from "./TopicsQueuesWkflw.js";
import { Prompter } from "@aws-doc-sdk-examples/lib/prompter.js";
import { SlowLogger } from "@aws-doc-sdk-examples/lib/slow-logger.js";

export const startSnsWorkflow = () => {
```

```
const noLoggerDelay = process.argv.find((arg) => arg === "--no-logger-delay");
const snsClient = new SNSClient({});
const sqsClient = new SQSClient({});
const prompter = new Prompter();
const logger = noLoggerDelay ? console : new SlowLogger(25);

const wkflw = new TopicsQueuesWkflw(snsClient, sqsClient, prompter, logger);

wkflw.start();
};
```

The preceding code provides the necessary dependencies and starts the workflow. The next section contains the bulk of the example.

```
const toneChoices = [
  { name: "cheerful", value: "cheerful" },
  { name: "funny", value: "funny" },
  { name: "serious", value: "serious" },
  { name: "sincere", value: "sincere" },
];

export class TopicsQueuesWkflw {
  // SNS topic is configured as First-In-First-Out
  isFifo = true;

  // Automatic content-based deduplication is enabled.
  autoDedup = false;

  snsClient;
  sqsClient;
  topicName;
  topicArn;
  subscriptionArns = [];
  /**
   * @type {{ queueName: string, queueArn: string, queueUrl: string, policy?:
string }[]}
   */
  queues = [];
  prompter;
```

```
/**
 * @param {import('@aws-sdk/client-sns').SNSClient} snsClient
 * @param {import('@aws-sdk/client-sqs').SQSClient} sqsClient
 * @param {import('../libs/prompter.js').Prompter} prompter
 * @param {import('../libs/logger.js').Logger} logger
 */
constructor(snsClient, sqsClient, prompter, logger) {
  this.snsClient = snsClient;
  this.sqsClient = sqsClient;
  this.prompter = prompter;
  this.logger = logger;
}

async welcome() {
  await this.logger.log(MESSAGES.description);
}

async confirmFifo() {
  await this.logger.log(MESSAGES.snsFifoDescription);
  this.isFifo = await this.prompter.confirm({
    message: MESSAGES.snsFifoPrompt,
  });

  if (this.isFifo) {
    this.logger.logSeparator(MESSAGES.headerDedup);
    await this.logger.log(MESSAGES.deduplicationNotice);
    await this.logger.log(MESSAGES.deduplicationDescription);
    this.autoDedup = await this.prompter.confirm({
      message: MESSAGES.deduplicationPrompt,
    });
  }
}

async createTopic() {
  await this.logger.log(MESSAGES.creatingTopics);
  this.topicName = await this.prompter.input({
    message: MESSAGES.topicNamePrompt,
  });
  if (this.isFifo) {
    this.topicName += ".fifo";
    this.logger.logSeparator(MESSAGES.headerFifoNaming);
    await this.logger.log(MESSAGES.appendFifoNotice);
  }
}
```

```
const response = await this.snsClient.send(
  new CreateTopicCommand({
    Name: this.topicName,
    Attributes: {
      FifoTopic: this.isFifo ? "true" : "false",
      ...(this.autoDedup ? { ContentBasedDeduplication: "true" } : {}),
    },
  }),
);

this.topicArn = response.TopicArn;

await this.logger.log(
  MESSAGES.topicCreatedNotice
    .replace("${TOPIC_NAME}", this.topicName)
    .replace("${TOPIC_ARN}", this.topicArn),
);
}

async createQueues() {
  await this.logger.log(MESSAGES.createQueuesNotice);
  // Increase this number to add more queues.
  let maxQueues = 2;

  for (let i = 0; i < maxQueues; i++) {
    await this.logger.log(MESSAGES.queueCount.replace("${COUNT}", i + 1));
    let queueName = await this.prompter.input({
      message: MESSAGES.queueNamePrompt.replace(
        "${EXAMPLE_NAME}",
        i === 0 ? "good-news" : "bad-news",
      ),
    },
  });

    if (this.isFifo) {
      queueName += ".fifo";
      await this.logger.log(MESSAGES.appendFifoNotice);
    }

    const response = await this.sqsClient.send(
      new CreateQueueCommand({
        QueueName: queueName,
        Attributes: { ...(this.isFifo ? { FifoQueue: "true" } : {} ) },
      }),
    );
```

```
const { Attributes } = await this.sqsClient.send(
  new GetQueueAttributesCommand({
    QueueUrl: response.QueueUrl,
    AttributeNames: ["QueueArn"],
  }),
);

this.queues.push({
  queueName,
  queueArn: Attributes.QueueArn,
  queueUrl: response.QueueUrl,
});

await this.logger.log(
  MESSAGES.queueCreatedNotice
    .replace("${QUEUE_NAME}", queueName)
    .replace("${QUEUE_URL}", response.QueueUrl)
    .replace("${QUEUE_ARN}", Attributes.QueueArn),
);
}
}

async attachQueueIamPolicies() {
  for (const [index, queue] of this.queues.entries()) {
    const policy = JSON.stringify(
      {
        Statement: [
          {
            Effect: "Allow",
            Principal: {
              Service: "sns.amazonaws.com",
            },
            Action: "sqs:SendMessage",
            Resource: queue.queueArn,
            Condition: {
              ArnEquals: {
                "aws:SourceArn": this.topicArn,
              },
            },
          },
        ],
      },
      null,
```

```
    2,
  );

  if (index !== 0) {
    this.logger.logSeparator();
  }

  await this.logger.log(MESSAGES.attachPolicyNotice);
  console.log(policy);
  const addPolicy = await this.prompter.confirm({
    message: MESSAGES.addPolicyConfirmation.replace(
      "${QUEUE_NAME}",
      queue.queueName,
    ),
  });

  if (addPolicy) {
    await this.sqsClient.send(
      new SetQueueAttributesCommand({
        QueueUrl: queue.queueUrl,
        Attributes: {
          Policy: policy,
        },
      }),
    );
    queue.policy = policy;
  } else {
    await this.logger.log(
      MESSAGES.policyNotAttachedNotice.replace(
        "${QUEUE_NAME}",
        queue.queueName,
      ),
    );
  }
}

async subscribeQueuesToTopic() {
  for (const [index, queue] of this.queues.entries()) {
    /**
     * @type {import('@aws-sdk/client-sns').SubscribeCommandInput}
     */
    const subscribeParams = {
      TopicArn: this.topicArn,
```

```
    Protocol: "sqs",
    Endpoint: queue.queueArn,
  };
  let tones = [];

  if (this.isFifo) {
    if (index === 0) {
      await this.logger.log(MESSAGES.fifoFilterNotice);
    }
    tones = await this.prompter.checkbox({
      message: MESSAGES.fifoFilterSelect.replace(
        "${QUEUE_NAME}",
        queue.queueName,
      ),
      choices: toneChoices,
    });

    if (tones.length) {
      subscribeParams.Attributes = {
        FilterPolicyScope: "MessageAttributes",
        FilterPolicy: JSON.stringify({
          tone: tones,
        }),
      };
    }
  }

  const { SubscriptionArn } = await this.snsClient.send(
    new SubscribeCommand(subscribeParams),
  );

  this.subscriptionArns.push(SubscriptionArn);

  await this.logger.log(
    MESSAGES.queueSubscribedNotice
      .replace("${QUEUE_NAME}", queue.queueName)
      .replace("${TOPIC_NAME}", this.topicName)
      .replace("${TONES}", tones.length ? tones.join(", ") : "none"),
  );
}

async publishMessages() {
  const message = await this.prompter.input({
```

```
    message: MESSAGES.publishMessagePrompt,
  });

  let groupId, deduplicationId, choices;

  if (this.isFifo) {
    await this.logger.log(MESSAGES.groupIdNotice);
    groupId = await this.prompter.input({
      message: MESSAGES.groupIdPrompt,
    });

    if (this.autoDedup === false) {
      await this.logger.log(MESSAGES.deduplicationIdNotice);
      deduplicationId = await this.prompter.input({
        message: MESSAGES.deduplicationIdPrompt,
      });
    }

    choices = await this.prompter.checkbox({
      message: MESSAGES.messageAttributesPrompt,
      choices: toneChoices,
    });
  }

  await this.snsClient.send(
    new PublishCommand({
      TopicArn: this.topicArn,
      Message: message,
      ...(groupId
        ? {
            MessageGroupId: groupId,
          }
        : {}),
      ...(deduplicationId
        ? {
            MessageDeduplicationId: deduplicationId,
          }
        : {}),
      ...(choices
        ? {
            MessageAttributes: {
              tone: {
                DataType: "String.Array",
                StringValue: JSON.stringify(choices),
              },
            },
          }
        : {}),
    })
  );
}
```



```
        },
      },
    }
    : {}),
  }),
);

const publishAnother = await this.prompter.confirm({
  message: MESSAGES.publishAnother,
});

if (publishAnother) {
  await this.publishMessages();
}
}

async receiveAndDeleteMessages() {
  for (const queue of this.queues) {
    const { Messages } = await this.sqsClient.send(
      new ReceiveMessageCommand({
        QueueUrl: queue.queueUrl,
      }),
    );

    if (Messages) {
      await this.logger.log(
        MESSAGES.messagesReceivedNotice.replace(
          "${QUEUE_NAME}",
          queue.queueName,
        ),
      );
      console.log(Messages);

      await this.sqsClient.send(
        new DeleteMessageBatchCommand({
          QueueUrl: queue.queueUrl,
          Entries: Messages.map((message) => ({
            Id: message.MessageId,
            ReceiptHandle: message.ReceiptHandle,
          })),
        }),
      );
    } else {
      await this.logger.log(
```

```
        MESSAGES.noMessagesReceivedNotice.replace(
            "${QUEUE_NAME}",
            queue.queueName,
        ),
    );
}
}

const deleteAndPoll = await this.prompter.confirm({
    message: MESSAGES.deleteAndPollConfirmation,
});

if (deleteAndPoll) {
    await this.receiveAndDeleteMessages();
}
}

async destroyResources() {
    for (const subscriptionArn of this.subscriptionArns) {
        await this.snsClient.send(
            new UnsubscribeCommand({ SubscriptionArn: subscriptionArn }),
        );
    }

    for (const queue of this.queues) {
        await this.sqsClient.send(
            new DeleteQueueCommand({ QueueUrl: queue.queueUrl }),
        );
    }

    if (this.topicArn) {
        await this.snsClient.send(
            new DeleteTopicCommand({ TopicArn: this.topicArn }),
        );
    }
}

async start() {
    console.clear();

    try {
        this.logger.logSeparator(MESSAGES.headerWelcome);
        await this.welcome();
        this.logger.logSeparator(MESSAGES.headerFifo);
    }
}
```

```
    await this.confirmFifo();
    this.logger.logSeparator(MESSAGES.headerCreateTopic);
    await this.createTopic();
    this.logger.logSeparator(MESSAGES.headerCreateQueues);
    await this.createQueues();
    this.logger.logSeparator(MESSAGES.headerAttachPolicy);
    await this.attachQueueIamPolicies();
    this.logger.logSeparator(MESSAGES.headerSubscribeQueues);
    await this.subscribeQueuesToTopic();
    this.logger.logSeparator(MESSAGES.headerPublishMessage);
    await this.publishMessages();
    this.logger.logSeparator(MESSAGES.headerReceiveMessages);
    await this.receiveAndDeleteMessages();
  } catch (err) {
    console.error(err);
  } finally {
    await this.destroyResources();
  }
}
```

- For API details, see the following topics in *AWS SDK for JavaScript API Reference*.
 - [CreateQueue](#)
 - [CreateTopic](#)
 - [DeleteMessageBatch](#)
 - [DeleteQueue](#)
 - [DeleteTopic](#)
 - [GetQueueAttributes](#)
 - [Publish](#)
 - [ReceiveMessage](#)
 - [SetQueueAttributes](#)
 - [Subscribe](#)
 - [Unsubscribe](#)

Serverless examples

Invoke a Lambda function from an Amazon SQS trigger

The following code example shows how to implement a Lambda function that receives an event triggered by receiving messages from an SQS queue. The function retrieves the messages from the event parameter and logs the content of each message.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [Serverless examples](#) repository.

Consuming an SQS event with Lambda using JavaScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const message of event.Records) {
    await processMessageAsync(message);
  }
  console.info("done");
};

async function processMessageAsync(message) {
  try {
    console.log(`Processed message ${message.body}`);
    // TODO: Do interesting work based on the new message
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

Consuming an SQS event with Lambda using TypeScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
```

```
// SPDX-License-Identifier: Apache-2.0
import { SQSEvent, Context, SQSHandler, SQSRecord } from "aws-lambda";

export const functionHandler: SQSHandler = async (
  event: SQSEvent,
  context: Context
): Promise<void> => {
  for (const message of event.Records) {
    await processMessageAsync(message);
  }
  console.info("done");
};

async function processMessageAsync(message: SQSRecord): Promise<any> {
  try {
    console.log(`Processed message ${message.body}`);
    // TODO: Do interesting work based on the new message
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

Reporting batch item failures for Lambda functions with an Amazon SQS trigger

The following code example shows how to implement partial batch response for Lambda functions that receive events from an SQS queue. The function reports the batch item failures in the response, signaling to Lambda to retry those messages later.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [Serverless examples](#) repository.

Reporting SQS batch item failures with Lambda using JavaScript.

```
// Node.js 20.x Lambda runtime, AWS SDK for Javascript V3
export const handler = async (event, context) => {
  const batchItemFailures = [];
  for (const record of event.Records) {
    try {
      await processMessageAsync(record, context);
    } catch (error) {
      batchItemFailures.push({ itemIdentifier: record.messageId });
    }
  }
  return { batchItemFailures };
};

async function processMessageAsync(record, context) {
  if (record.body && record.body.includes("error")) {
    throw new Error("There is an error in the SQS Message.");
  }
  console.log(`Processed message: ${record.body}`);
}
```

Reporting SQS batch item failures with Lambda using TypeScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SQSEvent, SQSBatchResponse, Context, SQSBatchItemFailure, SQSRecord } from
  'aws-lambda';

export const handler = async (event: SQSEvent, context: Context):
  Promise<SQSBatchResponse> => {
  const batchItemFailures: SQSBatchItemFailure[] = [];

  for (const record of event.Records) {
    try {
      await processMessageAsync(record);
    } catch (error) {
      batchItemFailures.push({ itemIdentifier: record.messageId });
    }
  }

  return {batchItemFailures: batchItemFailures};
};

async function processMessageAsync(record: SQSRecord): Promise<void> {
```

```
if (record.body && record.body.includes("error")) {
    throw new Error('There is an error in the SQS Message.');
```

```
}
```

```
console.log(`Processed message ${record.body}`);
```

```
}
```

Step Functions examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with Step Functions.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Topics

- [Actions](#)

Actions

StartExecution

The following code example shows how to use `StartExecution`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { SFNClient, StartExecutionCommand } from "@aws-sdk/client-sfn";

/**
```

```
* @param {{ sfncClient: SFNClient, stateMachineArn: string }} config
*/
export async function startExecution({ sfncClient, stateMachineArn }) {
  const response = await sfncClient.send(
    new StartExecutionCommand({
      stateMachineArn,
    }),
  );
  console.log(response);
  // Example response:
  // {
  //   '$metadata': {
  //     httpStatusCode: 200,
  //     requestId: '202a9309-c16a-454b-adeb-c4d19afe3bf2',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  //   executionArn: 'arn:aws:states:us-
east-1:000000000000:execution:MyStateMachine:aaaaaaaa-f787-49fb-a20c-1b61c64eafe6',
  //   startDate: 2024-01-04T15:54:08.362Z
  // }
  return response;
}

// Call function if run directly
import { fileURLToPath } from "url";
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  startExecution({ sfncClient: new SFNClient({}), stateMachineArn: "ARN" });
}
```

- For API details, see [StartExecution](#) in *AWS SDK for JavaScript API Reference*.

AWS STS examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with AWS STS.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Topics

- [Actions](#)

Actions

AssumeRole

The following code example shows how to use AssumeRole.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create the client.

```
import { STSClient } from "@aws-sdk/client-sts";
// Set the AWS Region.
const REGION = "us-east-1";
// Create an AWS STS service client object.
export const client = new STSClient({ region: REGION });
```

Assume the IAM role.

```
import { AssumeRoleCommand } from "@aws-sdk/client-sts";

import { client } from "../libs/client.js";

export const main = async () => {
  try {
    // Returns a set of temporary security credentials that you can use to
    // access Amazon Web Services resources that you might not normally
    // have access to.
    const command = new AssumeRoleCommand({
```

```
// The Amazon Resource Name (ARN) of the role to assume.
RoleArn: "ROLE_ARN",
// An identifier for the assumed role session.
RoleSessionName: "session1",
// The duration, in seconds, of the role session. The value specified
// can range from 900 seconds (15 minutes) up to the maximum session
// duration set for the role.
DurationSeconds: 900,
});
const response = await client.send(command);
console.log(response);
} catch (err) {
  console.error(err);
}
};
```

- For API details, see [AssumeRole](#) in *AWS SDK for JavaScript API Reference*.

SDK for JavaScript (v2)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

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

var roleToAssume = {
  RoleArn: "arn:aws:iam::123456789012:role/RoleName",
  RoleSessionName: "session1",
  DurationSeconds: 900,
};
var roleCreds;

// Create the STS service object
var sts = new AWS.STS({ apiVersion: "2011-06-15" });

//Assume Role
```

```
sts.assumeRole(roleToAssume, function (err, data) {
  if (err) console.log(err, err.stack);
  else {
    roleCreds = {
      accessKeyId: data.Credentials.AccessKeyId,
      secretAccessKey: data.Credentials.SecretAccessKey,
      sessionToken: data.Credentials.SessionToken,
    };
    stsGetCallerIdentity(roleCreds);
  }
});

//Get Arn of current identity
function stsGetCallerIdentity(creds) {
  var stsParams = { credentials: creds };
  // Create STS service object
  var sts = new AWS.STS(stsParams);

  sts.getCallerIdentity({}, function (err, data) {
    if (err) {
      console.log(err, err.stack);
    } else {
      console.log(data.Arn);
    }
  });
}
```

- For API details, see [AssumeRole](#) in *AWS SDK for JavaScript API Reference*.

AWS Support examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with AWS Support.

Basics are code examples that show you how to perform the essential operations within a service.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Get started

Hello AWS Support

The following code examples show how to get started using AWS Support.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Invoke `main()` to run the example.

```
import {
  DescribeServicesCommand,
  SupportClient,
} from "@aws-sdk/client-support";

// Change the value of 'region' to your preferred AWS Region.
const client = new SupportClient({ region: "us-east-1" });

const getServiceCount = async () => {
  try {
    const { services } = await client.send(new DescribeServicesCommand({}));
    return services.length;
  } catch (err) {
    if (err.name === "SubscriptionRequiredException") {
      throw new Error(
        "You must be subscribed to the AWS Support plan to use this feature.",
      );
    } else {
      throw err;
    }
  }
};

export const main = async () => {
  try {
    const count = await getServiceCount();
    console.log(`Hello, AWS Support! There are ${count} services available.`);
  }
};
```

```
    } catch (err) {  
      console.error("Failed to get service count: ", err.message);  
    }  
  };  
};
```

- For API details, see [DescribeServices](#) in *AWS SDK for JavaScript API Reference*.

Topics

- [Basics](#)
- [Actions](#)

Basics

Learn the basics

The following code example shows how to:

- Get and display available services and severity levels for cases.
- Create a support case using a selected service, category, and severity level.
- Get and display a list of open cases for the current day.
- Add an attachment set and a communication to the new case.
- Describe the new attachment and communication for the case.
- Resolve the case.
- Get and display a list of resolved cases for the current day.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Run an interactive scenario in the terminal.

```
import {
```

```
AddAttachmentsToSetCommand,
AddCommunicationToCaseCommand,
CreateCaseCommand,
DescribeAttachmentCommand,
DescribeCasesCommand,
DescribeCommunicationsCommand,
DescribeServicesCommand,
DescribeSeverityLevelsCommand,
ResolveCaseCommand,
SupportClient,
} from "@aws-sdk/client-support";
import * as inquirer from "@inquirer/prompts";
import { retry } from "@aws-doc-sdk-examples/lib/utils/util-timers.js";

const wrapText = (text, char = "=") => {
  const rule = char.repeat(80);
  return `${rule}\n  ${text}\n${rule}\n`;
};

const client = new SupportClient({ region: "us-east-1" });

// Verify that the account has a Support plan.
export const verifyAccount = async () => {
  const command = new DescribeServicesCommand({});

  try {
    await client.send(command);
  } catch (err) {
    if (err.name === "SubscriptionRequiredException") {
      throw new Error(
        "You must be subscribed to the AWS Support plan to use this feature.",
      );
    } else {
      throw err;
    }
  }
};

/**
 * Select a service from the list returned from DescribeServices.
 */
export const getService = async () => {
  const { services } = await client.send(new DescribeServicesCommand({}));
  const selectedService = await inquirer.select({
```

```
    message:
      "Select a service. Your support case will be created for this service. The
      list of services is truncated for readability.",
    choices: services.slice(0, 10).map((s) => ({ name: s.name, value: s })),
  });
  return selectedService;
};

/**
 * @param {{ categories: import('@aws-sdk/client-support').Category[]}} service
 */
export const getCategory = async (service) => {
  const selectedCategory = await inquirer.select({
    message: "Select a category.",
    choices: service.categories.map((c) => ({ name: c.name, value: c })),
  });
  return selectedCategory;
};

// Get the available severity levels for the account.
export const getSeverityLevel = async () => {
  const command = new DescribeSeverityLevelsCommand({});
  const { severityLevels } = await client.send(command);
  const selectedSeverityLevel = await inquirer.select({
    message: "Select a severity level.",
    choices: severityLevels.map((s) => ({ name: s.name, value: s })),
  });
  return selectedSeverityLevel;
};

/**
 * Create a new support case
 * @param {{
 *   selectedService: import('@aws-sdk/client-support').Service
 *   selectedCategory: import('@aws-sdk/client-support').Category
 *   selectedSeverityLevel: import('@aws-sdk/client-support').SeverityLevel
 * }} selections
 * @returns
 */
export const createCase = async ({
  selectedService,
  selectedCategory,
  selectedSeverityLevel,
}) => {
```

```
const command = new CreateCaseCommand({
  subject: "IGNORE: Test case",
  communicationBody: "This is a test. Please ignore.",
  serviceCode: selectedService.code,
  categoryCode: selectedCategory.code,
  severityCode: selectedSeverityLevel.code,
});
const { caseId } = await client.send(command);
return caseId;
};

// Get a list of open support cases created today.
export const getTodaysOpenCases = async () => {
  const d = new Date();
  const startOfToday = new Date(d.getFullYear(), d.getMonth(), d.getDate());
  const command = new DescribeCasesCommand({
    includeCommunications: false,
    afterTime: startOfToday.toISOString(),
  });

  const { cases } = await client.send(command);

  if (cases.length === 0) {
    throw new Error(
      "Unexpected number of cases. Expected more than 0 open cases.",
    );
  }
  return cases;
};

// Create an attachment set.
export const createAttachmentSet = async () => {
  const command = new AddAttachmentsToSetCommand({
    attachments: [
      {
        fileName: "example.txt",
        data: new TextEncoder().encode("some example text"),
      },
    ],
  });
  const { attachmentSetId } = await client.send(command);
  return attachmentSetId;
};
```



```
export const linkAttachmentSetToCase = async (attachmentSetId, caseId) => {
  const command = new AddCommunicationToCaseCommand({
    attachmentSetId,
    caseId,
    communicationBody: "Adding attachment set to case.",
  });
  await client.send(command);
};

// Get all communications for a support case.
export const getCommunications = async (caseId) => {
  const command = new DescribeCommunicationsCommand({
    caseId,
  });
  const { communications } = await client.send(command);
  return communications;
};

/**
 * @param {import('@aws-sdk/client-support').Communication[]} communications
 */
export const getFirstAttachment = (communications) => {
  const firstCommWithAttachment = communications.find(
    (c) => c.attachmentSet.length > 0,
  );
  return firstCommWithAttachment?.attachmentSet[0].attachmentId;
};

// Get an attachment.
export const getAttachment = async (attachmentId) => {
  const command = new DescribeAttachmentCommand({
    attachmentId,
  });
  const { attachment } = await client.send(command);
  return attachment;
};

// Resolve the case matching the given case ID.
export const resolveCase = async (caseId) => {
  const shouldResolve = await inquirer.confirm({
    message: `Do you want to resolve ${caseId}?`,
  });

  if (shouldResolve) {
```

```
    const command = new ResolveCaseCommand({
      caseId: caseId,
    });

    await client.send(command);
    return true;
  }
  return false;
};

/**
 * Find a specific case in the list of provided cases by case ID.
 * If the case is not found, and the results are paginated, continue
 * paging through the results.
 * @param {{
 *   caseId: string,
 *   cases: import('@aws-sdk/client-support').CaseDetails[]
 *   nextToken: string
 * }} options
 * @returns
 */
export const findCase = async ({ caseId, cases, nextToken }) => {
  const foundCase = cases.find((c) => c.caseId === caseId);

  if (foundCase) {
    return foundCase;
  }

  if (nextToken) {
    const response = await client.send(
      new DescribeCasesCommand({
        nextToken,
        includeResolvedCases: true,
      }),
    );
    return findCase({
      caseId,
      cases: response.cases,
      nextToken: response.nextToken,
    });
  }

  throw new Error(`${caseId} not found.`);
};
```

```
// Get all cases created today.
export const getTodaysResolvedCases = async (caseIdToWaitFor) => {
  const d = new Date("2023-01-18");
  const startOfToday = new Date(d.getFullYear(), d.getMonth(), d.getDate());
  const command = new DescribeCasesCommand({
    includeCommunications: false,
    afterTime: startOfToday.toISOString(),
    includeResolvedCases: true,
  });
  const { cases, nextToken } = await client.send(command);
  await findCase({ cases, caseId: caseIdToWaitFor, nextToken });
  return cases.filter((c) => c.status === "resolved");
};

const main = async () => {
  let caseId;
  try {
    console.log(wrapText("Welcome to the AWS Support basic usage scenario."));

    // Verify that the account is subscribed to support.
    await verifyAccount();

    // Provided a truncated list of services and prompt the user to select one.
    const selectedService = await getService();

    // Provided the categories for the selected service and prompt the user to
select one.
    const selectedCategory = await getCategory(selectedService);

    // Provide the severity available severity levels for the account and prompt the
user to select one.
    const selectedSeverityLevel = await getSeverityLevel();

    // Create a support case.
    console.log("\nCreating a support case.");
    caseId = await createCase({
      selectedService,
      selectedCategory,
      selectedSeverityLevel,
    });
    console.log(`Support case created: ${caseId}`);

    // Display a list of open support cases created today.
```

```
const todaysOpenCases = await retry(
  { intervalInMs: 1000, maxRetries: 15 },
  getTodaysOpenCases,
);
console.log(
  `\\n0open support cases created today: ${todaysOpenCases.length}`,
);
console.log(todaysOpenCases.map((c) => `${c.caseId}`).join("\\n"));

// Create an attachment set.
console.log("\\nCreating an attachment set.");
const attachmentSetId = await createAttachmentSet();
console.log(`Attachment set created: ${attachmentSetId}`);

// Add the attachment set to the support case.
console.log(`\\nAdding attachment set to ${caseId}`);
await linkAttachmentSetToCase(attachmentSetId, caseId);
console.log(`Attachment set added to ${caseId}`);

// List the communications for a support case.
console.log(`\\nListing communications for ${caseId}`);
const communications = await getCommunications(caseId);
console.log(
  communications
    .map(
      (c) =>
        `Communication created on ${c.timeCreated}. Has
        ${c.attachmentSet.length} attachments.`
    )
    .join("\\n"),
);

// Describe the first attachment.
console.log(`\\nDescribing attachment ${attachmentSetId}`);
const attachmentId = getFirstAttachment(communications);
const attachment = await getAttachment(attachmentId);
console.log(
  `Attachment is the file '${
    attachment.fileName
  }' with data: \\n${new TextDecoder().decode(attachment.data)}`,
);

// Confirm that the support case should be resolved.
const isResolved = await resolveCase(caseId);
```

```
if (isResolved) {
  // List the resolved cases and include the one previously created.
  // Resolved cases can take a while to appear.
  console.log(
    "\nWaiting for case status to be marked as resolved. This can take some
time.",
  );
  const resolvedCases = await retry(
    { intervalInMs: 20000, maxRetries: 15 },
    () => getTodayResolvedCases(caseId),
  );
  console.log("Resolved cases:");
  console.log(resolvedCases.map((c) => c.caseId).join("\n"));
}
} catch (err) {
  console.error(err);
}
};
```

- For API details, see the following topics in *AWS SDK for JavaScript API Reference*.
 - [AddAttachmentsToSet](#)
 - [AddCommunicationToCase](#)
 - [CreateCase](#)
 - [DescribeAttachment](#)
 - [DescribeCases](#)
 - [DescribeCommunications](#)
 - [DescribeServices](#)
 - [DescribeSeverityLevels](#)
 - [ResolveCase](#)

Actions

AddAttachmentsToSet

The following code example shows how to use AddAttachmentsToSet.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { AddAttachmentsToSetCommand } from "@aws-sdk/client-support";

import { client } from "../libs/client.js";

export const main = async () => {
  try {
    // Create a new attachment set or add attachments to an existing set.
    // Provide an 'attachmentSetId' value to add attachments to an existing set.
    // Use AddCommunicationToCase or CreateCase to associate an attachment set with
    a support case.
    const response = await client.send(
      new AddAttachmentsToSetCommand({
        // You can add up to three attachments per set. The size limit is 5 MB per
        attachment.
        attachments: [
          {
            fileName: "example.txt",
            data: new TextEncoder().encode("some example text"),
          },
        ],
      })),
    );
    // Use this ID in AddCommunicationToCase or CreateCase.
    console.log(response.attachmentSetId);
    return response;
  } catch (err) {
    console.error(err);
  }
};
```

- For API details, see [AddAttachmentsToSet](#) in *AWS SDK for JavaScript API Reference*.

AddCommunicationToCase

The following code example shows how to use AddCommunicationToCase.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { AddCommunicationToCaseCommand } from "@aws-sdk/client-support";

import { client } from "../libs/client.js";

export const main = async () => {
  let attachmentSetId;

  try {
    // Add a communication to a case.
    const response = await client.send(
      new AddCommunicationToCaseCommand({
        communicationBody: "Adding an attachment.",
        // Set value to an existing support case id.
        caseId: "CASE_ID",
        // Optional. Set value to an existing attachment set id to add attachments
        // to the case.
        attachmentSetId,
      }),
    );
    console.log(response);
    return response;
  } catch (err) {
    console.error(err);
  }
};
```

- For API details, see [AddCommunicationToCase](#) in *AWS SDK for JavaScript API Reference*.

CreateCase

The following code example shows how to use CreateCase.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { CreateCaseCommand } from "@aws-sdk/client-support";

import { client } from "../libs/client.js";

export const main = async () => {
  try {
    // Create a new case and log the case id.
    // Important: This creates a real support case in your account.
    const response = await client.send(
      new CreateCaseCommand({
        // The subject line of the case.
        subject: "IGNORE: Test case",
        // Use DescribeServices to find available service codes for each service.
        serviceCode: "service-quicksight-end-user",
        // Use DescribeSecurityLevels to find available severity codes for your
        support plan.
        severityCode: "low",
        // Use DescribeServices to find available category codes for each service.
        categoryCode: "end-user-support",
        // The main description of the support case.
        communicationBody: "This is a test. Please ignore.",
      }),
    );
    console.log(response.caseId);
    return response;
  } catch (err) {
    console.error(err);
  }
};
```


- For API details, see [CreateCase](#) in *AWS SDK for JavaScript API Reference*.

DescribeAttachment

The following code example shows how to use DescribeAttachment.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { DescribeAttachmentCommand } from "@aws-sdk/client-support";

import { client } from "../libs/client.js";

export const main = async () => {
  try {
    // Get the metadata and content of an attachment.
    const response = await client.send(
      new DescribeAttachmentCommand({
        // Set value to an existing attachment id.
        // Use DescribeCommunications or DescribeCases to find an attachment id.
        attachmentId: "ATTACHMENT_ID",
      }),
    );
    console.log(response.attachment?.fileName);
    return response;
  } catch (err) {
    console.error(err);
  }
};
```

- For API details, see [DescribeAttachment](#) in *AWS SDK for JavaScript API Reference*.

DescribeCases

The following code example shows how to use DescribeCases.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { DescribeCasesCommand } from "@aws-sdk/client-support";

import { client } from "../libs/client.js";

export const main = async () => {
  try {
    // Get all of the unresolved cases in your account.
    // Filter or expand results by providing parameters to the DescribeCasesCommand.
    Refer
    // to the TypeScript definition and the API doc for more information on possible
    parameters.
    // https://docs.aws.amazon.com/AWSJavaScriptSDK/v3/latest/clients/client-
    support/interfaces/describecasescommandinput.html
    const response = await client.send(new DescribeCasesCommand({}));
    const caseIds = response.cases.map((supportCase) => supportCase.caseId);
    console.log(caseIds);
    return response;
  } catch (err) {
    console.error(err);
  }
};
```

- For API details, see [DescribeCases](#) in *AWS SDK for JavaScript API Reference*.

DescribeCommunications

The following code example shows how to use `DescribeCommunications`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { DescribeCommunicationsCommand } from "@aws-sdk/client-support";

import { client } from "../libs/client.js";

export const main = async () => {
  try {
    // Get all communications for the support case.
    // Filter results by providing parameters to the DescribeCommunicationsCommand.
    Refer
    // to the TypeScript definition and the API doc for more information on possible
    parameters.
    // https://docs.aws.amazon.com/AWSJavaScriptSDK/v3/latest/clients/client-
    support/interfaces/describecommunicationscommandinput.html
    const response = await client.send(
      new DescribeCommunicationsCommand({
        // Set value to an existing case id.
        caseId: "CASE_ID",
      }),
    );
    const text = response.communications.map((item) => item.body).join("\n");
    console.log(text);
    return response;
  } catch (err) {
    console.error(err);
  }
};
```

- For API details, see [DescribeCommunications](#) in *AWS SDK for JavaScript API Reference*.

DescribeSeverityLevels

The following code example shows how to use `DescribeSeverityLevels`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { DescribeSeverityLevelsCommand } from "@aws-sdk/client-support";

import { client } from "../libs/client.js";

export const main = async () => {
  try {
    // Get the list of severity levels.
    // The available values depend on the support plan for the account.
    const response = await client.send(new DescribeSeverityLevelsCommand({}));
    console.log(response.severityLevels);
    return response;
  } catch (err) {
    console.error(err);
  }
};
```

- For API details, see [DescribeSeverityLevels](#) in *AWS SDK for JavaScript API Reference*.

ResolveCase

The following code example shows how to use `ResolveCase`.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
import { ResolveCaseCommand } from "@aws-sdk/client-support";
```

```
import { client } from "../libs/client.js";

const main = async () => {
  try {
    const response = await client.send(
      new ResolveCaseCommand({
        caseId: "CASE_ID",
      }),
    );

    console.log(response.finalCaseStatus);
    return response;
  } catch (err) {
    console.error(err);
  }
};
```

- For API details, see [ResolveCase](#) in *AWS SDK for JavaScript API Reference*.

Amazon Textract examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with Amazon Textract.

Scenarios are code examples that show you how to accomplish specific tasks by calling multiple functions within a service or combined with other AWS services.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Topics

- [Scenarios](#)

Scenarios

Create an Amazon Textract explorer application

The following code example shows how to explore Amazon Textract output through an interactive application.

SDK for JavaScript (v3)

Shows how to use the AWS SDK for JavaScript to build a React application that uses Amazon Textract to extract data from a document image and display it in an interactive web page. This example runs in a web browser and requires an authenticated Amazon Cognito identity for credentials. It uses Amazon Simple Storage Service (Amazon S3) for storage, and for notifications it polls an Amazon Simple Queue Service (Amazon SQS) queue that is subscribed to an Amazon Simple Notification Service (Amazon SNS) topic.

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- Amazon Cognito Identity
- Amazon S3
- Amazon SNS
- Amazon SQS
- Amazon Textract

Create an application to analyze customer feedback

The following code example shows how to create an application that analyzes customer comment cards, translates them from their original language, determines their sentiment, and generates an audio file from the translated text.

SDK for JavaScript (v3)

This example application analyzes and stores customer feedback cards. Specifically, it fulfills the need of a fictitious hotel in New York City. The hotel receives feedback from guests in various languages in the form of physical comment cards. That feedback is uploaded into the app through a web client. After an image of a comment card is uploaded, the following steps occur:

- Text is extracted from the image using Amazon Textract.
- Amazon Comprehend determines the sentiment of the extracted text and its language.
- The extracted text is translated to English using Amazon Translate.
- Amazon Polly synthesizes an audio file from the extracted text.

The full app can be deployed with the AWS CDK. For source code and deployment instructions, see the project in [GitHub](#). The following excerpts show how the AWS SDK for JavaScript is used inside of Lambda functions.

```
import {
  ComprehendClient,
  DetectDominantLanguageCommand,
  DetectSentimentCommand,
} from "@aws-sdk/client-comprehend";

/**
 * Determine the language and sentiment of the extracted text.
 *
 * @param {{ source_text: string }} extractTextOutput
 */
export const handler = async (extractTextOutput) => {
  const comprehendClient = new ComprehendClient({});

  const detectDominantLanguageCommand = new DetectDominantLanguageCommand({
    Text: extractTextOutput.source_text,
  });

  // The source language is required for sentiment analysis and
  // translation in the next step.
  const { Languages } = await comprehendClient.send(
    detectDominantLanguageCommand,
  );

  const languageCode = Languages[0].LanguageCode;

  const detectSentimentCommand = new DetectSentimentCommand({
    Text: extractTextOutput.source_text,
    LanguageCode: languageCode,
  });

  const { Sentiment } = await comprehendClient.send(detectSentimentCommand);

  return {
    sentiment: Sentiment,
    language_code: languageCode,
  };
};
```

```
import {
  DetectDocumentTextCommand,
  TextractClient,
} from "@aws-sdk/client-textract";

/**
 * Fetch the S3 object from the event and analyze it using Amazon Textract.
 *
 * @param {import("@types/aws-lambda").EventBridgeEvent<"Object Created">}
  eventBridgeS3Event
 */
export const handler = async (eventBridgeS3Event) => {
  const textractClient = new TextractClient();

  const detectDocumentTextCommand = new DetectDocumentTextCommand({
    Document: {
      S3Object: {
        Bucket: eventBridgeS3Event.bucket,
        Name: eventBridgeS3Event.object,
      },
    },
  });

  // Textract returns a list of blocks. A block can be a line, a page, word, etc.
  // Each block also contains geometry of the detected text.
  // For more information on the Block type, see https://docs.aws.amazon.com/textract/latest/dg/API\_Block.html.
  const { Blocks } = await textractClient.send(detectDocumentTextCommand);

  // For the purpose of this example, we are only interested in words.
  const extractedWords = Blocks.filter((b) => b.BlockType === "WORD").map(
    (b) => b.Text,
  );

  return extractedWords.join(" ");
};
```

```
import { PollyClient, SynthesizeSpeechCommand } from "@aws-sdk/client-polly";
import { S3Client } from "@aws-sdk/client-s3";
import { Upload } from "@aws-sdk/lib-storage";

/**
 * Synthesize an audio file from text.
```



```
*
* @param {{ bucket: string, translated_text: string, object: string}}
sourceDestinationConfig
*/
export const handler = async (sourceDestinationConfig) => {
  const pollyClient = new PollyClient({});

  const synthesizeSpeechCommand = new SynthesizeSpeechCommand({
    Engine: "neural",
    Text: sourceDestinationConfig.translated_text,
    VoiceId: "Ruth",
    OutputFormat: "mp3",
  });

  const { AudioStream } = await pollyClient.send(synthesizeSpeechCommand);

  const audioKey = `${sourceDestinationConfig.object}.mp3`;

  // Store the audio file in S3.
  const s3Client = new S3Client();
  const upload = new Upload({
    client: s3Client,
    params: {
      Bucket: sourceDestinationConfig.bucket,
      Key: audioKey,
      Body: AudioStream,
      ContentType: "audio/mp3",
    },
  });

  await upload.done();
  return audioKey;
};
```

```
import {
  TranslateClient,
  TranslateTextCommand,
} from "@aws-sdk/client-translate";

/**
 * Translate the extracted text to English.
 *
```

```
* @param {{ extracted_text: string, source_language_code: string }}
textAndSourceLanguage
*/
export const handler = async (textAndSourceLanguage) => {
  const translateClient = new TranslateClient({});

  const translateCommand = new TranslateTextCommand({
    SourceLanguageCode: textAndSourceLanguage.source_language_code,
    TargetLanguageCode: "en",
    Text: textAndSourceLanguage.extracted_text,
  });

  const { TranslatedText } = await translateClient.send(translateCommand);

  return { translated_text: TranslatedText };
};
```

Services used in this example

- Amazon Comprehend
- Lambda
- Amazon Polly
- Amazon Textract
- Amazon Translate

Amazon Transcribe examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with Amazon Transcribe.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Scenarios are code examples that show you how to accomplish specific tasks by calling multiple functions within a service or combined with other AWS services.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Topics

- [Actions](#)
- [Scenarios](#)

Actions

DeleteMedicalTranscriptionJob

The following code example shows how to use DeleteMedicalTranscriptionJob.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create the client.

```
import { TranscribeClient } from "@aws-sdk/client-transcribe";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon Transcribe service client object.
const transcribeClient = new TranscribeClient({ region: REGION });
export { transcribeClient };
```

Delete a medical transcription job.

```
// Import the required AWS SDK clients and commands for Node.js
import { DeleteMedicalTranscriptionJobCommand } from "@aws-sdk/client-transcribe";
import { transcribeClient } from "../libs/transcribeClient.js";

// Set the parameters
export const params = {
  MedicalTranscriptionJobName: "MEDICAL_JOB_NAME", // For example,
  'medical_transcription_demo'
};

export const run = async () => {
```

```
try {
  const data = await transcribeClient.send(
    new DeleteMedicalTranscriptionJobCommand(params)
  );
  console.log("Success - deleted");
  return data; // For unit tests.
} catch (err) {
  console.log("Error", err);
}
};
run();
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [DeleteMedicalTranscriptionJob](#) in *AWS SDK for JavaScript API Reference*.

DeleteTranscriptionJob

The following code example shows how to use DeleteTranscriptionJob.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Delete a transcription job.

```
// Import the required AWS SDK clients and commands for Node.js
import { DeleteTranscriptionJobCommand } from "@aws-sdk/client-transcribe";
import { transcribeClient } from "../libs/transcribeClient.js";

// Set the parameters
export const params = {
  TranscriptionJobName: "JOB_NAME", // Required. For example, 'transcription_demo'
};

export const run = async () => {
```

```
try {
  const data = await transcribeClient.send(
    new DeleteTranscriptionJobCommand(params)
  );
  console.log("Success - deleted");
  return data; // For unit tests.
} catch (err) {
  console.log("Error", err);
}
};
run();
```

Create the client.

```
import { TranscribeClient } from "@aws-sdk/client-transcribe";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon Transcribe service client object.
const transcribeClient = new TranscribeClient({ region: REGION });
export { transcribeClient };
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [DeleteTranscriptionJob](#) in *AWS SDK for JavaScript API Reference*.

ListMedicalTranscriptionJobs

The following code example shows how to use ListMedicalTranscriptionJobs.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create the client.

```
import { TranscribeClient } from "@aws-sdk/client-transcribe";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon Transcribe service client object.
const transcribeClient = new TranscribeClient({ region: REGION });
export { transcribeClient };
```

List medical transcription jobs.

```
// Import the required AWS SDK clients and commands for Node.js
import { StartMedicalTranscriptionJobCommand } from "@aws-sdk/client-transcribe";
import { transcribeClient } from "../libs/transcribeClient.js";

// Set the parameters
export const params = {
  MedicalTranscriptionJobName: "MEDICAL_JOB_NAME", // Required
  OutputBucketName: "OUTPUT_BUCKET_NAME", // Required
  Specialty: "PRIMARYCARE", // Required. Possible values are 'PRIMARYCARE'
  Type: "JOB_TYPE", // Required. Possible values are 'CONVERSATION' and 'DICTATION'
  LanguageCode: "LANGUAGE_CODE", // For example, 'en-US'
  MediaFormat: "SOURCE_FILE_FORMAT", // For example, 'wav'
  Media: {
    MediaFileUri: "SOURCE_FILE_LOCATION",
    // The S3 object location of the input media file. The URI must be in the same
    region
    // as the API endpoint that you are calling. For example,
    // "https://transcribe-demo.s3-REGION.amazonaws.com/hello_world.wav"
  },
};

export const run = async () => {
  try {
    const data = await transcribeClient.send(
      new StartMedicalTranscriptionJobCommand(params)
    );
    console.log("Success - put", data);
    return data; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};

run();
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [ListMedicalTranscriptionJobs](#) in *AWS SDK for JavaScript API Reference*.

ListTranscriptionJobs

The following code example shows how to use ListTranscriptionJobs.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

List transcription jobs.

```
// Import the required AWS SDK clients and commands for Node.js

import { ListTranscriptionJobsCommand } from "@aws-sdk/client-transcribe";
import { transcribeClient } from "../libs/transcribeClient.js";

// Set the parameters
export const params = {
  JobNameContains: "KEYWORD", // Not required. Returns only transcription
  // job names containing this string
};

export const run = async () => {
  try {
    const data = await transcribeClient.send(
      new ListTranscriptionJobsCommand(params)
    );
    console.log("Success", data.TranscriptionJobSummaries);
    return data; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
```

```
run();
```

Create the client.

```
import { TranscribeClient } from "@aws-sdk/client-transcribe";  
// Set the AWS Region.  
const REGION = "REGION"; //e.g. "us-east-1"  
// Create an Amazon Transcribe service client object.  
const transcribeClient = new TranscribeClient({ region: REGION });  
export { transcribeClient };
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [ListTranscriptionJobs](#) in *AWS SDK for JavaScript API Reference*.

StartMedicalTranscriptionJob

The following code example shows how to use StartMedicalTranscriptionJob.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Create the client.

```
import { TranscribeClient } from "@aws-sdk/client-transcribe";  
// Set the AWS Region.  
const REGION = "REGION"; //e.g. "us-east-1"  
// Create an Amazon Transcribe service client object.  
const transcribeClient = new TranscribeClient({ region: REGION });  
export { transcribeClient };
```

Start a medical transcription job.


```
// Import the required AWS SDK clients and commands for Node.js
import { StartMedicalTranscriptionJobCommand } from "@aws-sdk/client-transcribe";
import { transcribeClient } from "../libs/transcribeClient.js";

// Set the parameters
export const params = {
  MedicalTranscriptionJobName: "MEDICAL_JOB_NAME", // Required
  OutputBucketName: "OUTPUT_BUCKET_NAME", // Required
  Specialty: "PRIMARYCARE", // Required. Possible values are 'PRIMARYCARE'
  Type: "JOB_TYPE", // Required. Possible values are 'CONVERSATION' and 'DICTATION'
  LanguageCode: "LANGUAGE_CODE", // For example, 'en-US'
  MediaFormat: "SOURCE_FILE_FORMAT", // For example, 'wav'
  Media: {
    MediaFileUri: "SOURCE_FILE_LOCATION",
    // The S3 object location of the input media file. The URI must be in the same
    // region
    // as the API endpoint that you are calling. For example,
    // "https://transcribe-demo.s3-REGION.amazonaws.com/hello_world.wav"
  },
};

export const run = async () => {
  try {
    const data = await transcribeClient.send(
      new StartMedicalTranscriptionJobCommand(params)
    );
    console.log("Success - put", data);
    return data; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};

run();
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [StartMedicalTranscriptionJob](#) in *AWS SDK for JavaScript API Reference*.

StartTranscriptionJob

The following code example shows how to use StartTranscriptionJob.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Start a transcription job.

```
// Import the required AWS SDK clients and commands for Node.js
import { StartTranscriptionJobCommand } from "@aws-sdk/client-transcribe";
import { transcribeClient } from "../libs/transcribeClient.js";

// Set the parameters
export const params = {
  TranscriptionJobName: "JOB_NAME",
  LanguageCode: "LANGUAGE_CODE", // For example, 'en-US'
  MediaFormat: "SOURCE_FILE_FORMAT", // For example, 'wav'
  Media: {
    MediaFileUri: "SOURCE_LOCATION",
    // For example, "https://transcribe-demo.s3-REGION.amazonaws.com/
hello_world.wav"
  },
  OutputBucketName: "OUTPUT_BUCKET_NAME"
};

export const run = async () => {
  try {
    const data = await transcribeClient.send(
      new StartTranscriptionJobCommand(params)
    );
    console.log("Success - put", data);
    return data; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

Create the client.

```
import { TranscribeClient } from "@aws-sdk/client-transcribe";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon Transcribe service client object.
const transcribeClient = new TranscribeClient({ region: REGION });
export { transcribeClient };
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [StartTranscriptionJob](#) in *AWS SDK for JavaScript API Reference*.

Scenarios

Build an Amazon Transcribe streaming app

The following code example shows how to build an app that records, transcribes, and translates live audio in real-time, and emails the results.

SDK for JavaScript (v3)

Shows how to use Amazon Transcribe to build an app that records, transcribes, and translates live audio in real-time, and emails the results using Amazon Simple Email Service (Amazon SES).

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- Amazon Comprehend
- Amazon SES
- Amazon Transcribe
- Amazon Translate

Amazon Translate examples using SDK for JavaScript (v3)

The following code examples show you how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with Amazon Translate.

Scenarios are code examples that show you how to accomplish specific tasks by calling multiple functions within a service or combined with other AWS services.

Each example includes a link to the complete source code, where you can find instructions on how to set up and run the code in context.

Topics

- [Scenarios](#)

Scenarios

Build an Amazon Transcribe streaming app

The following code example shows how to build an app that records, transcribes, and translates live audio in real-time, and emails the results.

SDK for JavaScript (v3)

Shows how to use Amazon Transcribe to build an app that records, transcribes, and translates live audio in real-time, and emails the results using Amazon Simple Email Service (Amazon SES).

For complete source code and instructions on how to set up and run, see the full example on [GitHub](#).

Services used in this example

- Amazon Comprehend
- Amazon SES
- Amazon Transcribe
- Amazon Translate

Building an Amazon Lex chatbot

The following code example shows how to create a chatbot to engage your website visitors.

SDK for JavaScript (v3)

Shows how to use the Amazon Lex API to create a Chatbot within a web application to engage your web site visitors.

For complete source code and instructions on how to set up and run, see the full example [Building an Amazon Lex chatbot](#) in the AWS SDK for JavaScript developer guide.

Services used in this example

- Amazon Comprehend
- Amazon Lex
- Amazon Translate

Create an application to analyze customer feedback

The following code example shows how to create an application that analyzes customer comment cards, translates them from their original language, determines their sentiment, and generates an audio file from the translated text.

SDK for JavaScript (v3)

This example application analyzes and stores customer feedback cards. Specifically, it fulfills the need of a fictitious hotel in New York City. The hotel receives feedback from guests in various languages in the form of physical comment cards. That feedback is uploaded into the app through a web client. After an image of a comment card is uploaded, the following steps occur:

- Text is extracted from the image using Amazon Textract.
- Amazon Comprehend determines the sentiment of the extracted text and its language.
- The extracted text is translated to English using Amazon Translate.
- Amazon Polly synthesizes an audio file from the extracted text.

The full app can be deployed with the AWS CDK. For source code and deployment instructions, see the project in [GitHub](#). The following excerpts show how the AWS SDK for JavaScript is used inside of Lambda functions.

```
import {
  ComprehendClient,
  DetectDominantLanguageCommand,
  DetectSentimentCommand,
} from "@aws-sdk/client-comprehend";

/**
 * Determine the language and sentiment of the extracted text.
 */
```

```
* @param {{ source_text: string}} extractTextOutput
*/
export const handler = async (extractTextOutput) => {
  const comprehendClient = new ComprehendClient({});

  const detectDominantLanguageCommand = new DetectDominantLanguageCommand({
    Text: extractTextOutput.source_text,
  });

  // The source language is required for sentiment analysis and
  // translation in the next step.
  const { Languages } = await comprehendClient.send(
    detectDominantLanguageCommand,
  );

  const languageCode = Languages[0].LanguageCode;

  const detectSentimentCommand = new DetectSentimentCommand({
    Text: extractTextOutput.source_text,
    LanguageCode: languageCode,
  });

  const { Sentiment } = await comprehendClient.send(detectSentimentCommand);

  return {
    sentiment: Sentiment,
    language_code: languageCode,
  };
};
```

```
import {
  DetectDocumentTextCommand,
  TextractClient,
} from "@aws-sdk/client-textract";

/**
 * Fetch the S3 object from the event and analyze it using Amazon Textract.
 *
 * @param {import("@types/aws-lambda").EventBridgeEvent<"Object Created">}
  eventBridgeS3Event
 */
export const handler = async (eventBridgeS3Event) => {
  const textractClient = new TextractClient();
```

```

const detectDocumentTextCommand = new DetectDocumentTextCommand({
  Document: {
    S3Object: {
      Bucket: eventBridgeS3Event.bucket,
      Name: eventBridgeS3Event.object,
    },
  },
});

// Textract returns a list of blocks. A block can be a line, a page, word, etc.
// Each block also contains geometry of the detected text.
// For more information on the Block type, see https://docs.aws.amazon.com/textract/latest/dg/API\_Block.html.
const { Blocks } = await textractClient.send(detectDocumentTextCommand);

// For the purpose of this example, we are only interested in words.
const extractedWords = Blocks.filter((b) => b.BlockType === "WORD").map(
  (b) => b.Text,
);

return extractedWords.join(" ");
};

```

```

import { PollyClient, SynthesizeSpeechCommand } from "@aws-sdk/client-polly";
import { S3Client } from "@aws-sdk/client-s3";
import { Upload } from "@aws-sdk/lib-storage";

/**
 * Synthesize an audio file from text.
 *
 * @param {{ bucket: string, translated_text: string, object: string }}
 * sourceDestinationConfig
 */
export const handler = async (sourceDestinationConfig) => {
  const pollyClient = new PollyClient({});

  const synthesizeSpeechCommand = new SynthesizeSpeechCommand({
    Engine: "neural",
    Text: sourceDestinationConfig.translated_text,
    VoiceId: "Ruth",
    OutputFormat: "mp3",
  });
};

```

```
const { AudioStream } = await pollyClient.send(synthesizeSpeechCommand);

const audioKey = `${sourceDestinationConfig.object}.mp3`;

// Store the audio file in S3.
const s3Client = new S3Client();
const upload = new Upload({
  client: s3Client,
  params: {
    Bucket: sourceDestinationConfig.bucket,
    Key: audioKey,
    Body: AudioStream,
    ContentType: "audio/mp3",
  },
});

await upload.done();
return audioKey;
};
```

```
import {
  TranslateClient,
  TranslateTextCommand,
} from "@aws-sdk/client-translate";

/**
 * Translate the extracted text to English.
 *
 * @param {{ extracted_text: string, source_language_code: string }}
  textAndSourceLanguage
 */
export const handler = async (textAndSourceLanguage) => {
  const translateClient = new TranslateClient({});

  const translateCommand = new TranslateTextCommand({
    SourceLanguageCode: textAndSourceLanguage.source_language_code,
    TargetLanguageCode: "en",
    Text: textAndSourceLanguage.extracted_text,
  });

  const { TranslatedText } = await translateClient.send(translateCommand);
```



```
    return { translated_text: TranslatedText };  
};
```

Services used in this example

- Amazon Comprehend
- Lambda
- Amazon Polly
- Amazon Textract
- Amazon Translate

Security for this AWS Product or Service

Cloud security at Amazon Web Services (AWS) is the highest priority. As an AWS customer, you benefit from a data center and network architecture that is built to meet the requirements of the most security-sensitive organizations. Security is a shared responsibility between AWS and you. The [Shared Responsibility Model](#) describes this as Security of the Cloud and Security in the Cloud.

Security of the Cloud – AWS is responsible for protecting the infrastructure that runs all of the services offered in the AWS Cloud and providing you with services that you can use securely. Our security responsibility is the highest priority at AWS, and the effectiveness of our security is regularly tested and verified by third-party auditors as part of the [AWS Compliance Programs](#).

Security in the Cloud – Your responsibility is determined by the AWS service you are using, and other factors including the sensitivity of your data, your organization's requirements, and applicable laws and regulations.

This AWS product or service follows the [shared responsibility model](#) through the specific Amazon Web Services (AWS) services it supports. For AWS service security information, see the [AWS service security documentation page](#) and [AWS services that are in scope of AWS compliance efforts by compliance program](#).

Topics

- [Data protection in this AWS product or service](#)
- [Identity and Access Management](#)
- [Compliance Validation for this AWS Product or Service](#)
- [Resilience for this AWS Product or Service](#)
- [Infrastructure Security for this AWS Product or Service](#)
- [Enforce a minimum TLS version](#)

Data protection in this AWS product or service

The AWS [shared responsibility model](#) applies to data protection in this AWS product or service. As described in this model, AWS is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this infrastructure. You are also responsible for the security configuration and management tasks

for the AWS services that you use. For more information about data privacy, see the [Data Privacy FAQ](#). For information about data protection in Europe, see the [AWS Shared Responsibility Model and GDPR](#) blog post on the *AWS Security Blog*.

For data protection purposes, we recommend that you protect AWS account credentials and set up individual users with AWS IAM Identity Center or AWS Identity and Access Management (IAM). That way, each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources. We require TLS 1.2 and recommend TLS 1.3.
- Set up API and user activity logging with AWS CloudTrail.
- Use AWS encryption solutions, along with all default security controls within AWS services.
- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing sensitive data that is stored in Amazon S3.
- If you require FIPS 140-3 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see [Federal Information Processing Standard \(FIPS\) 140-3](#).

We strongly recommend that you never put confidential or sensitive information, such as your customers' email addresses, into tags or free-form text fields such as a **Name** field. This includes when you work with this AWS product or service or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into tags or free-form text fields used for names may be used for billing or diagnostic logs. If you provide a URL to an external server, we strongly recommend that you do not include credentials information in the URL to validate your request to that server.

Identity and Access Management

AWS Identity and Access Management (IAM) is an AWS service that helps an administrator securely control access to AWS resources. IAM administrators control who can be *authenticated* (signed in) and *authorized* (have permissions) to use AWS resources. IAM is an AWS service that you can use with no additional charge.

Topics

- [Audience](#)

- [Authenticating with identities](#)
- [Managing access using policies](#)
- [How AWS services work with IAM](#)
- [Troubleshooting AWS identity and access](#)

Audience

How you use AWS Identity and Access Management (IAM) differs, depending on the work that you do in AWS.

Service user – If you use AWS services to do your job, then your administrator provides you with the credentials and permissions that you need. As you use more AWS features to do your work, you might need additional permissions. Understanding how access is managed can help you request the right permissions from your administrator. If you cannot access a feature in AWS, see [Troubleshooting AWS identity and access](#) or the user guide of the AWS service you are using.

Service administrator – If you're in charge of AWS resources at your company, you probably have full access to AWS. It's your job to determine which AWS features and resources your service users should access. You must then submit requests to your IAM administrator to change the permissions of your service users. Review the information on this page to understand the basic concepts of IAM. To learn more about how your company can use IAM with AWS, see the user guide of the AWS service you are using.

IAM administrator – If you're an IAM administrator, you might want to learn details about how you can write policies to manage access to AWS. To view example AWS identity-based policies that you can use in IAM, see the user guide of the AWS service you are using.

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. You must be *authenticated* (signed in to AWS) as the AWS account root user, as an IAM user, or by assuming an IAM role.

You can sign in to AWS as a federated identity by using credentials provided through an identity source. AWS IAM Identity Center (IAM Identity Center) users, your company's single sign-on authentication, and your Google or Facebook credentials are examples of federated identities. When you sign in as a federated identity, your administrator previously set up identity federation using IAM roles. When you access AWS by using federation, you are indirectly assuming a role.

Depending on the type of user you are, you can sign in to the AWS Management Console or the AWS access portal. For more information about signing in to AWS, see [How to sign in to your AWS account](#) in the *AWS Sign-In User Guide*.

If you access AWS programmatically, AWS provides a software development kit (SDK) and a command line interface (CLI) to cryptographically sign your requests by using your credentials. If you don't use AWS tools, you must sign requests yourself. For more information about using the recommended method to sign requests yourself, see [Signing AWS API requests](#) in the *IAM User Guide*.

Regardless of the authentication method that you use, you might be required to provide additional security information. For example, AWS recommends that you use multi-factor authentication (MFA) to increase the security of your account. To learn more, see [Multi-factor authentication](#) in the *AWS IAM Identity Center User Guide* and [Using multi-factor authentication \(MFA\) in AWS](#) in the *IAM User Guide*.

AWS account root user

When you create an AWS account, you begin with one sign-in identity that has complete access to all AWS services and resources in the account. This identity is called the AWS account *root user* and is accessed by signing in with the email address and password that you used to create the account. We strongly recommend that you don't use the root user for your everyday tasks. Safeguard your root user credentials and use them to perform the tasks that only the root user can perform. For the complete list of tasks that require you to sign in as the root user, see [Tasks that require root user credentials](#) in the *IAM User Guide*.

Federated identity

As a best practice, require human users, including users that require administrator access, to use federation with an identity provider to access AWS services by using temporary credentials.

A *federated identity* is a user from your enterprise user directory, a web identity provider, the AWS Directory Service, the Identity Center directory, or any user that accesses AWS services by using credentials provided through an identity source. When federated identities access AWS accounts, they assume roles, and the roles provide temporary credentials.

For centralized access management, we recommend that you use AWS IAM Identity Center. You can create users and groups in IAM Identity Center, or you can connect and synchronize to a set of users and groups in your own identity source for use across all your AWS accounts and applications. For

information about IAM Identity Center, see [What is IAM Identity Center?](#) in the *AWS IAM Identity Center User Guide*.

IAM users and groups

An [IAM user](#) is an identity within your AWS account that has specific permissions for a single person or application. Where possible, we recommend relying on temporary credentials instead of creating IAM users who have long-term credentials such as passwords and access keys. However, if you have specific use cases that require long-term credentials with IAM users, we recommend that you rotate access keys. For more information, see [Rotate access keys regularly for use cases that require long-term credentials](#) in the *IAM User Guide*.

An [IAM group](#) is an identity that specifies a collection of IAM users. You can't sign in as a group. You can use groups to specify permissions for multiple users at a time. Groups make permissions easier to manage for large sets of users. For example, you could have a group named *IAMAdmins* and give that group permissions to administer IAM resources.

Users are different from roles. A user is uniquely associated with one person or application, but a role is intended to be assumable by anyone who needs it. Users have permanent long-term credentials, but roles provide temporary credentials. To learn more, see [When to create an IAM user \(instead of a role\)](#) in the *IAM User Guide*.

IAM roles

An [IAM role](#) is an identity within your AWS account that has specific permissions. It is similar to an IAM user, but is not associated with a specific person. You can temporarily assume an IAM role in the AWS Management Console by [switching roles](#). You can assume a role by calling an AWS CLI or AWS API operation or by using a custom URL. For more information about methods for using roles, see [Using IAM roles](#) in the *IAM User Guide*.

IAM roles with temporary credentials are useful in the following situations:

- **Federated user access** – To assign permissions to a federated identity, you create a role and define permissions for the role. When a federated identity authenticates, the identity is associated with the role and is granted the permissions that are defined by the role. For information about roles for federation, see [Creating a role for a third-party Identity Provider](#) in the *IAM User Guide*. If you use IAM Identity Center, you configure a permission set. To control what your identities can access after they authenticate, IAM Identity Center correlates the permission set to a role in IAM. For information about permission sets, see [Permission sets](#) in the *AWS IAM Identity Center User Guide*.

- **Temporary IAM user permissions** – An IAM user or role can assume an IAM role to temporarily take on different permissions for a specific task.
- **Cross-account access** – You can use an IAM role to allow someone (a trusted principal) in a different account to access resources in your account. Roles are the primary way to grant cross-account access. However, with some AWS services, you can attach a policy directly to a resource (instead of using a role as a proxy). To learn the difference between roles and resource-based policies for cross-account access, see [Cross account resource access in IAM](#) in the *IAM User Guide*.
- **Cross-service access** – Some AWS services use features in other AWS services. For example, when you make a call in a service, it's common for that service to run applications in Amazon EC2 or store objects in Amazon S3. A service might do this using the calling principal's permissions, using a service role, or using a service-linked role.
 - **Forward access sessions (FAS)** – When you use an IAM user or role to perform actions in AWS, you are considered a principal. When you use some services, you might perform an action that then initiates another action in a different service. FAS uses the permissions of the principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. FAS requests are only made when a service receives a request that requires interactions with other AWS services or resources to complete. In this case, you must have permissions to perform both actions. For policy details when making FAS requests, see [Forward access sessions](#).
 - **Service role** – A service role is an [IAM role](#) that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see [Creating a role to delegate permissions to an AWS service](#) in the *IAM User Guide*.
 - **Service-linked role** – A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.
- **Applications running on Amazon EC2** – You can use an IAM role to manage temporary credentials for applications that are running on an EC2 instance and making AWS CLI or AWS API requests. This is preferable to storing access keys within the EC2 instance. To assign an AWS role to an EC2 instance and make it available to all of its applications, you create an instance profile that is attached to the instance. An instance profile contains the role and enables programs that are running on the EC2 instance to get temporary credentials. For more information, see [Using an IAM role to grant permissions to applications running on Amazon EC2 instances](#) in the *IAM User Guide*.

To learn whether to use IAM roles or IAM users, see [When to create an IAM role \(instead of a user\)](#) in the *IAM User Guide*.

Managing access using policies

You control access in AWS by creating policies and attaching them to AWS identities or resources. A policy is an object in AWS that, when associated with an identity or resource, defines their permissions. AWS evaluates these policies when a principal (user, root user, or role session) makes a request. Permissions in the policies determine whether the request is allowed or denied. Most policies are stored in AWS as JSON documents. For more information about the structure and contents of JSON policy documents, see [Overview of JSON policies](#) in the *IAM User Guide*.

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

By default, users and roles have no permissions. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies. The administrator can then add the IAM policies to roles, and users can assume the roles.

IAM policies define permissions for an action regardless of the method that you use to perform the operation. For example, suppose that you have a policy that allows the `iam:GetRole` action. A user with that policy can get role information from the AWS Management Console, the AWS CLI, or the AWS API.

Identity-based policies

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see [Creating IAM policies](#) in the *IAM User Guide*.

Identity-based policies can be further categorized as *inline policies* or *managed policies*. Inline policies are embedded directly into a single user, group, or role. Managed policies are standalone policies that you can attach to multiple users, groups, and roles in your AWS account. Managed policies include AWS managed policies and customer managed policies. To learn how to choose between a managed policy or an inline policy, see [Choosing between managed policies and inline policies](#) in the *IAM User Guide*.

Resource-based policies

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are *IAM role trust policies* and *Amazon S3 bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must [specify a principal](#) in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

Resource-based policies are inline policies that are located in that service. You can't use AWS managed policies from IAM in a resource-based policy.

Access control lists (ACLs)

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

Amazon S3, AWS WAF, and Amazon VPC are examples of services that support ACLs. To learn more about ACLs, see [Access control list \(ACL\) overview](#) in the *Amazon Simple Storage Service Developer Guide*.

Other policy types

AWS supports additional, less-common policy types. These policy types can set the maximum permissions granted to you by the more common policy types.

- **Permissions boundaries** – A permissions boundary is an advanced feature in which you set the maximum permissions that an identity-based policy can grant to an IAM entity (IAM user or role). You can set a permissions boundary for an entity. The resulting permissions are the intersection of an entity's identity-based policies and its permissions boundaries. Resource-based policies that specify the user or role in the `Principal` field are not limited by the permissions boundary. An explicit deny in any of these policies overrides the allow. For more information about permissions boundaries, see [Permissions boundaries for IAM entities](#) in the *IAM User Guide*.
- **Service control policies (SCPs)** – SCPs are JSON policies that specify the maximum permissions for an organization or organizational unit (OU) in AWS Organizations. AWS Organizations is a service for grouping and centrally managing multiple AWS accounts that your business owns. If you enable all features in an organization, then you can apply service control policies (SCPs) to

any or all of your accounts. The SCP limits permissions for entities in member accounts, including each AWS account root user. For more information about Organizations and SCPs, see [Service control policies](#) in the *AWS Organizations User Guide*.

- **Session policies** – Session policies are advanced policies that you pass as a parameter when you programmatically create a temporary session for a role or federated user. The resulting session's permissions are the intersection of the user or role's identity-based policies and the session policies. Permissions can also come from a resource-based policy. An explicit deny in any of these policies overrides the allow. For more information, see [Session policies](#) in the *IAM User Guide*.

Multiple policy types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see [Policy evaluation logic](#) in the *IAM User Guide*.

How AWS services work with IAM

To get a high-level view of how AWS services work with most IAM features, see [AWS services that work with IAM](#) in the *IAM User Guide*.

To learn how to use a specific AWS service with IAM, see the security section of the relevant service's User Guide.

Troubleshooting AWS identity and access

Use the following information to help you diagnose and fix common issues that you might encounter when working with AWS and IAM.

Topics

- [I am not authorized to perform an action in AWS](#)
- [I am not authorized to perform iam:PassRole](#)
- [I want to allow people outside of my AWS account to access my AWS resources](#)

I am not authorized to perform an action in AWS

If you receive an error that you're not authorized to perform an action, your policies must be updated to allow you to perform the action.

The following example error occurs when the `mateojackson` IAM user tries to use the console to view details about a fictional `my-example-widget` resource but doesn't have the fictional `awes:GetWidget` permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
awes:GetWidget on resource: my-example-widget
```

In this case, the policy for the `mateojackson` user must be updated to allow access to the `my-example-widget` resource by using the `awes:GetWidget` action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I am not authorized to perform iam:PassRole

If you receive an error that you're not authorized to perform the `iam:PassRole` action, your policies must be updated to allow you to pass a role to AWS.

Some AWS services allow you to pass an existing role to that service instead of creating a new service role or service-linked role. To do this, you must have permissions to pass the role to the service.

The following example error occurs when an IAM user named `marymajor` tries to use the console to perform an action in AWS. However, the action requires the service to have permissions that are granted by a service role. Mary does not have permissions to pass the role to the service.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
iam:PassRole
```

In this case, Mary's policies must be updated to allow her to perform the `iam:PassRole` action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I want to allow people outside of my AWS account to access my AWS resources

You can create a role that users in other accounts or people outside of your organization can use to access your resources. You can specify who is trusted to assume the role. For services that support resource-based policies or access control lists (ACLs), you can use those policies to grant people access to your resources.

To learn more, consult the following:

- To learn whether AWS supports these features, see [How AWS services work with IAM](#).
- To learn how to provide access to your resources across AWS accounts that you own, see [Providing access to an IAM user in another AWS account that you own](#) in the *IAM User Guide*.
- To learn how to provide access to your resources to third-party AWS accounts, see [Providing access to AWS accounts owned by third parties](#) in the *IAM User Guide*.
- To learn how to provide access through identity federation, see [Providing access to externally authenticated users \(identity federation\)](#) in the *IAM User Guide*.
- To learn the difference between using roles and resource-based policies for cross-account access, see [Cross account resource access in IAM](#) in the *IAM User Guide*.

Compliance Validation for this AWS Product or Service

To learn whether an AWS service is within the scope of specific compliance programs, see [AWS services in Scope by Compliance Program](#) and choose the compliance program that you are interested in. For general information, see [AWS Compliance Programs](#).

You can download third-party audit reports using AWS Artifact. For more information, see [Downloading Reports in AWS Artifact](#).

Your compliance responsibility when using AWS services is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. AWS provides the following resources to help with compliance:

- [Security and Compliance Quick Start Guides](#) – These deployment guides discuss architectural considerations and provide steps for deploying baseline environments on AWS that are security and compliance focused.
- [Architecting for HIPAA Security and Compliance on Amazon Web Services](#) – This whitepaper describes how companies can use AWS to create HIPAA-eligible applications.

Note

Not all AWS services are HIPAA eligible. For more information, see the [HIPAA Eligible Services Reference](#).

- [AWS Compliance Resources](#) – This collection of workbooks and guides might apply to your industry and location.
- [AWS Customer Compliance Guides](#) – Understand the shared responsibility model through the lens of compliance. The guides summarize the best practices for securing AWS services and map the guidance to security controls across multiple frameworks (including National Institute of Standards and Technology (NIST), Payment Card Industry Security Standards Council (PCI), and International Organization for Standardization (ISO)).
- [Evaluating Resources with Rules](#) in the *AWS Config Developer Guide* – The AWS Config service assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.
- [AWS Security Hub](#) – This AWS service provides a comprehensive view of your security state within AWS. Security Hub uses security controls to evaluate your AWS resources and to check your compliance against security industry standards and best practices. For a list of supported services and controls, see [Security Hub controls reference](#).
- [Amazon GuardDuty](#) – This AWS service detects potential threats to your AWS accounts, workloads, containers, and data by monitoring your environment for suspicious and malicious activities. GuardDuty can help you address various compliance requirements, like PCI DSS, by meeting intrusion detection requirements mandated by certain compliance frameworks.
- [AWS Audit Manager](#) – This AWS service helps you continuously audit your AWS usage to simplify how you manage risk and compliance with regulations and industry standards.

This AWS product or service follows the [shared responsibility model](#) through the specific Amazon Web Services (AWS) services it supports. For AWS service security information, see the [AWS service security documentation page](#) and [AWS services that are in scope of AWS compliance efforts by compliance program](#).

Resilience for this AWS Product or Service

The AWS global infrastructure is built around AWS Regions and Availability Zones.

AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking.

With Availability Zones, you can design and operate applications and databases that automatically fail over between zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information about AWS Regions and Availability Zones, see [AWS Global Infrastructure](#).

This AWS product or service follows the [shared responsibility model](#) through the specific Amazon Web Services (AWS) services it supports. For AWS service security information, see the [AWS service security documentation page](#) and [AWS services that are in scope of AWS compliance efforts by compliance program](#).

Infrastructure Security for this AWS Product or Service

This AWS product or service uses managed services, and therefore is protected by the AWS global network security. For information about AWS security services and how AWS protects infrastructure, see [AWS Cloud Security](#). To design your AWS environment using the best practices for infrastructure security, see [Infrastructure Protection](#) in *Security Pillar AWS Well-Architected Framework*.

You use AWS published API calls to access this AWS Product or Service through the network. Clients must support the following:

- Transport Layer Security (TLS). We require TLS 1.2 and recommend TLS 1.3.
- Cipher suites with perfect forward secrecy (PFS) such as DHE (Ephemeral Diffie-Hellman) or ECDHE (Elliptic Curve Ephemeral Diffie-Hellman). Most modern systems such as Java 7 and later support these modes.

Additionally, requests must be signed by using an access key ID and a secret access key that is associated with an IAM principal. Or you can use the [AWS Security Token Service](#) (AWS STS) to generate temporary security credentials to sign requests.

This AWS product or service follows the [shared responsibility model](#) through the specific Amazon Web Services (AWS) services it supports. For AWS service security information, see the [AWS service security documentation page](#) and [AWS services that are in scope of AWS compliance efforts by compliance program](#).

Enforce a minimum TLS version

To add increased security when communicating with AWS services, configure the AWS SDK for JavaScript to use TLS 1.2 or later.

⚠ Important

The AWS SDK for JavaScript v3 automatically negotiates the highest level TLS version supported by a given AWS Service endpoint. You can optionally enforce a minimum TLS version required by your application, such as TLS 1.2 or 1.3, but please note that TLS 1.3 is not supported by some AWS Service endpoints, so some calls may fail if you enforce TLS 1.3.

Transport Layer Security (TLS) is a protocol used by web browsers and other applications to ensure the privacy and integrity of data exchanged over a network.

Verify and enforce TLS in Node.js

When you use the AWS SDK for JavaScript with Node.js, the underlying Node.js security layer is used to set the TLS version.

Node.js 12.0.0 and later use a minimum version of OpenSSL 1.1.1b, which supports TLS 1.3. The AWS SDK for JavaScript v3 defaults to use TLS 1.3 when available, but defaults to a lower version if required.

Verify the version of OpenSSL and TLS

To get the version of OpenSSL used by Node.js on your computer, run the following command.

```
node -p process.versions
```

The version of OpenSSL in the list is the version used by Node.js, as shown in the following example.

```
openssl: '1.1.1b'
```

To get the version of TLS used by Node.js on your computer, start the Node shell and run the following commands, in order.

```
> var tls = require("tls");  
> var tlsSocket = new tls.TLSSocket();  
> tlsSocket.getProtocol();
```

The last command outputs the TLS version, as shown in the following example.

```
'TLSv1.3'
```

Node.js defaults to use this version of TLS, and tries to negotiate another version of TLS if a call is not successful.

Enforce a minimum version of TLS

Node.js negotiates a version of TLS when a call fails. You can enforce the minimum allowable TLS version during this negotiation, either when running a script from the command line or per request in your JavaScript code.

To specify the minimum TLS version from the command line, you must use Node.js version 11.0.0 or later. To install a specific Node.js version, first install Node Version Manager (nvm) using the steps found at [Node version manager installing and updating](#). Then run the following commands to install and use a specific version of Node.js.

```
nvm install 11
nvm use 11
```

Enforce TLS 1.2

To enforce that TLS 1.2 is the minimum allowable version, specify the `--tls-min-v1.2` argument when running your script, as shown in the following example.

```
node --tls-min-v1.2 yourScript.js
```

To specify the minimum allowable TLS version for a specific request in your JavaScript code, use the `httpOptions` parameter to specify the protocol, as shown in the following example.

```
import https from "https";
import { NodeHttpHandler } from "@smithy/node-http-handler";
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({
  region: "us-west-2",
  requestHandler: new NodeHttpHandler({
    httpsAgent: new https.Agent(
      {
        secureProtocol: 'TLSv1_2_method'
      }
    )
  })
});
```



```
        }  
      )  
    })  
  });
```

Enforce TLS 1.3

To enforce that TLS 1.3 is the minimum allowable version, specify the `--tls-min-v1.3` argument when running your script, as shown in the following example.

```
node --tls-min-v1.3 yourScript.js
```

To specify the minimum allowable TLS version for a specific request in your JavaScript code, use the `httpOptions` parameter to specify the protocol, as shown in the following example.

```
import https from "https";  
import { NodeHttpHandler } from "@smithy/node-http-handler";  
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";  
  
const client = new DynamoDBClient({  
  region: "us-west-2",  
  requestHandler: new NodeHttpHandler({  
    httpsAgent: new https.Agent(  
      {  
        secureProtocol: 'TLSv1_3_method'  
      }  
    )  
  })  
});
```

Verify and enforce TLS in a browser script

When you use the SDK for JavaScript in a browser script, browser settings control the version of TLS that is used. The version of TLS used by the browser cannot be discovered or set by script and must be configured by the user. To verify and enforce the version of TLS used in a browser script, refer to the instructions for your specific browser.

Microsoft Internet Explorer

1. Open **Internet Explorer**.

2. From the menu bar, choose **Tools - Internet Options - Advanced** tab.
3. Scroll down to **Security** category, manually check the option box for **Use TLS 1.2**.
4. Click **OK**.
5. Close your browser and restart Internet Explorer.

Microsoft Edge

1. In the Windows menu search box, type *Internet options*.
2. Under **Best match**, click **Internet Options**.
3. In the **Internet Properties** window, on the **Advanced** tab, scroll down to the **Security** section.
4. Check the **User TLS 1.2** checkbox.
5. Click **OK**.

Google Chrome

1. Open **Google Chrome**.
2. Click **Alt F** and select **Settings**.
3. Scroll down and select **Show advanced settings...**
4. Scroll down to the **System** section and click on **Open proxy settings...**
5. Select the **Advanced** tab.
6. Scroll down to **Security** category, manually check the option box for **Use TLS 1.2**.
7. Click **OK**.
8. Close your browser and restart Google Chrome.

Mozilla Firefox

1. Open **Firefox**.
2. In the address bar, type **about:config** and press Enter.
3. In the **Search** field, enter **tls**. Find and double-click the entry for **security.tls.version.min**.
4. Set the integer value to 3 to force protocol of TLS 1.2 to be the default.
5. Click **OK**.
6. Close your browser and restart Mozilla Firefox.

Apple Safari

There are no options for enabling SSL protocols. If you are using Safari version 7 or greater, TLS 1.2 is automatically enabled.

Migrate from version 2.x to 3.x of the AWS SDK for JavaScript

The AWS SDK for JavaScript version 3 is a major rewrite of version 2. The section describes the differences between the two versions and explains how to migrate from version 2 to version 3 of the SDK for JavaScript.

Migrate your code to SDK for JavaScript v3 using codemod

AWS SDK for JavaScript version 3 (v3) comes with modernized interfaces for client configurations and utilities, which include credentials, Amazon S3 multipart upload, DynamoDB document client, waiters, and more. You can find what changed in v2 and the v3 equivalents for each change in the [migration guide on the AWS SDK for JavaScript GitHub repo](#).

To take full advantage of the AWS SDK for JavaScript v3, we recommend using the codemod scripts described below.

Use codemod to migrate existing v2 code

The collection of codemod scripts in [aws-sdk-js-codemod](#) helps migrate your existing AWS SDK for JavaScript (v2) application to use v3 APIs. You can run the transform as follows.

```
$ npx aws-sdk-js-codemod -t v2-to-v3 PATH...
```

For example, consider you have the following code, which creates a Amazon DynamoDB client from v2 and calls `listTables` operation.

```
// example.ts
import AWS from "aws-sdk";

const region = "us-west-2";
const client = new AWS.DynamoDB({ region });
await client.listTables({}).promise()
  .then(console.log)
  .catch(console.error);
```

You can run our `v2-to-v3` transform on `example.ts` as follows.

```
$ npx aws-sdk-js-codemod -t v2-to-v3 example.ts
```

The transform will convert the DynamoDB import to v3, create v3 client and call the `listTables` operation as follows.

```
// example.ts
import { DynamoDB } from "@aws-sdk/client-dynamodb";

const region = "us-west-2";
const client = new DynamoDB({ region });
await client.listTables()
  .then(console.log)
  .catch(console.error);
```

We've implemented transforms for common use cases. If your code doesn't transform correctly, please create a [bug report](#) or [feature request](#) with example input code and observed/expected output code. If your specific use case is already reported in [an existing issue](#), show your support by an upvote.

What's new in Version 3

Version 3 of the SDK for JavaScript (v3) contains the following new features.

Modularized packages

Users can now use a separate package for each service.

New middleware stack

Users can now use a middleware stack to control the lifecycle of an operation call.

In addition, the SDK is written in TypeScript, which has many advantages, such as static typing.

Important

The code examples for v3 in this guide are written in ECMAScript 6 (ES6). ES6 brings new syntax and new features to make your code more modern and readable, and do more. ES6 requires you use Node.js version 13.x or higher. To download and install the latest version

of Node.js, see [Node.js downloads](#). For more information, see [JavaScript ES6/CommonJS syntax](#).

Modularized packages

Version 2 of the SDK for JavaScript (v2) required you to use the entire AWS SDK, as follows.

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

Loading the entire SDK isn't an issue if your application is using many AWS services. However, if you need to use only a few AWS services, it means increasing the size of your application with code you don't need or use.

In v3, you can load and use only the individual AWS Services you need. This is shown in the following example, which gives you access to Amazon DynamoDB (DynamoDB).

```
import { DynamoDB } from "@aws-sdk/client-dynamodb";
```

Not only can you load and use individual AWS services, but you can also load and use only the service commands you need. This is shown in the following examples, which gives you access to DynamoDB client and the `ListTablesCommand` command.

```
import {
  DynamoDBClient,
  ListTablesCommand
} from "@aws-sdk/client-dynamodb";
```

Important

You should not import submodules into modules. For example, the following code might result in errors.

```
import { CognitoIdentity } from "@aws-sdk/client-cognito-identity/
CognitoIdentity";
```

The following is the correct code.

```
import { CognitoIdentity } from "@aws-sdk/client-cognito-identity";
```

Comparing code size

In Version 2 (v2), a simple code example that lists all of your Amazon DynamoDB tables in the us-west-2 Region might look like the following.

```
var AWS = require("aws-sdk");
// Set the Region
AWS.config.update({ region: "us-west-2" });
// Create 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("Tables names are ", data.TableNames);
  }
});
```

v3 looks like the following.

```
import {
  DynamoDBClient,
  ListTablesCommand
} from "@aws-sdk/client-dynamodb";

const dbclient = new DynamoDBClient({ region: "us-west-2" });

try {
  const results = await dbclient.send(new ListTablesCommand);

  for (const item of results.TableNames) {
    console.log(item);
  }
} catch (err) {
  console.error(err)
```

```
}
```

The `aws-sdk` package adds about 40 MB to your application. Replacing `var AWS = require("aws-sdk")` with `import {DynamoDB} from "@aws-sdk/client-dynamodb"` reduces that overhead to about 3 MB. Restricting the import to just the DynamoDB client and `ListTablesCommand` command reduces the overhead to less than 100 KB.

```
// Load the DynamoDB client and ListTablesCommand command for Node.js
import {
  DynamoDBClient,
  ListTablesCommand
} from "@aws-sdk/client-dynamodb";
const dbclient = new DynamoDBClient({});
```

Calling commands in v3

You can perform operations in v3 using either v2 or v3 commands. To use v3 commands you import the commands and the required AWS Services package clients, and run the command using the `.send` method using the `async/await` pattern.

To use v2 commands you import the required AWS Services packages, and run the v2 command directly in the package using either a callback or `async/await` pattern.

Using v3 commands

v3 provides a set of commands for each AWS Service package to enable you to perform operations for that AWS Service. After you install an AWS Service, you can browse the available commands in your project's `node-modules/@aws-sdk/client-PACKAGE_NAME/commands` folder.

You must import the commands you want to use. For example, the following code loads the DynamoDB service, and the `CreateTableCommand` command.

```
import { DynamoDB, CreateTableCommand } from "@aws-sdk/client-dynamodb";
```

To call these commands in the recommended `async/await` pattern, use the following syntax.

```
CLIENT.send(new XXXCommand);
```

For example, the following example creates a DynamoDB table using the recommended `async/await` pattern.


```
import { DynamoDB, CreateTableCommand } from "@aws-sdk/client-dynamodb";
const dynamodb = new DynamoDB({ region: "us-west-2" });
const tableParams = {
  TableName: TABLE_NAME
};

try {
  const data = await dynamodb.send(new CreateTableCommand(tableParams));
  console.log("Success", data);
} catch (err) {
  console.log("Error", err);
};
```

Using v2 commands

To use v2 commands in the SDK for JavaScript, you import the full AWS Service packages, as demonstrated in the following code.

```
const { DynamoDB } = require('@aws-sdk/client-dynamodb');
```

To call v2 commands in the recommended async/await pattern, use the following syntax.

```
client.command(parameters);
```

The following example uses the v2 `createTable` command to create a DynamoDB table using the recommended async/await pattern.

```
const { DynamoDB } = require('@aws-sdk/client-dynamodb');
const dynamoDB = new DynamoDB({ region: 'us-west-2' });
var tableParams = {
  TableName: TABLE_NAME
};
async function run() => {
  try {
    const data = await dynamoDB.createTable(tableParams);
    console.log("Success", data);
  }
  catch (err) {
    console.log("Error", err);
  }
};
```

```
run());
```

The following example uses the v2 `createBucket` command to create an Amazon S3 bucket using the callback pattern.

```
const { S3 } = require('@aws-sdk/client-s3');
const s3 = new S3({ region: 'us-west-2' });
var bucketParams = {
  Bucket : BUCKET_NAME
};
function run() {
  s3.createBucket(bucketParams, function (err, data) {
    if (err) {
      console.log("Error", err);
    } else {
      console.log("Success", data.Location);
    }
  })
};
run();
```

New middleware stack

v2 of the SDK enabled you to modify a request throughout the multiple stages of its lifecycle by attaching event listeners to the request. This approach can make it difficult to debug what went wrong during a request's lifecycle.

In v3, you can use a new middleware stack to control the lifecycle of an operation call. This approach provides a couple of benefits. Each middleware stage in the stack calls the next middleware stage after making any changes to the request object. This also makes debugging issues in the stack much easier, because you can see exactly which middleware stages were called leading up to the error.

The following example adds a custom header to a Amazon DynamoDB client (which we created and showed earlier) using middleware. The first argument is a function that accepts `next`, which is the next middleware stage in the stack to call, and `context`, which is an object that contains some information about the operation being called. The function returns a function that accepts `args`, which is an object that contains the parameters passed to the operation and the request. It returns the result from calling the next middleware with `args`.

```
dbclient.middlewareStack.add(  
  (next, context) => args => {  
    args.request.headers["Custom-Header"] = "value";  
    return next(args);  
  },  
  {  
    name: "my-middleware",  
    override: true,  
    step: "build"  
  }  
);  
  
dbclient.send(new PutObjectCommand(params));
```

What's different between the AWS SDK for JavaScript v2 and v3

This section captures notable changes from AWS SDK for JavaScript v2 to v3. Because v3 is a modular rewrite of v2, some basic concepts are different between v2 and v3. You can learn about these changes in our [blog posts](#). The following blog posts will get you up to speed:

- [Modular packages in AWS SDK for JavaScript](#)
- [Introducing Middleware Stack in Modular AWS SDK for JavaScript](#)

The summary of interface changes from AWS SDK for JavaScript v2 to v3 is given below. The goal is to help you easily find the v3 equivalents of the v2 APIs you are already familiar with.

Topics

- [Client constructors](#)
- [Credential providers](#)
- [Amazon S3 considerations](#)
- [DynamoDB document client](#)
- [Waiters and signers](#)
- [Notes on specific service clients](#)

Client constructors

This list is indexed by [v2 config parameters](#).

- [computeChecksums](#)
 - **v2:** Whether to compute MD5 checksums for payload bodies when the service accepts it (currently supported in S3 only).
 - **v3:** applicable commands of S3 (PutObject, PutBucketCors, etc.) will automatically compute the MD5 checksums for of the request payload. You can also specify a different checksum algorithm in the commands' `ChecksumAlgorithm` parameter to use a different checksum algorithm. You can find more information in the [S3 feature announcement](#).
- [convertResponseTypes](#)
 - **v2:** Whether types are converted when parsing response data.
 - **v3: Deprecated.** This option is considered not type-safe because it doesn't convert the types like time stamp or base64 binaries from the JSON response.
- [correctClockSkew](#)
 - **v2:** Whether to apply a clock skew correction and retry requests that fail because of an skewed client clock.
 - **v3: Deprecated.** SDK *always* applies a clock skew correction.
- [systemClockOffset](#)
 - **v2:** An offset value in milliseconds to apply to all signing times.
 - **v3:** No change.
- [credentials](#)
 - **v2:** The AWS credentials to sign requests with.
 - **v3:** No change. It can also be an async function that returns credentials. If the function returns an `expiration` (`Date`), the function will be called again when the expiration datetime nears. See [v3 API reference for `AwsAuthInputConfig` credentials](#).
- [endpointCacheSize](#)
 - **v2:** The size of the global cache storing endpoints from endpoint discovery operations.
 - **v3:** No change.
- [endpointDiscoveryEnabled](#)
 - **v2:** Whether to call operations with endpoints given by service dynamically.
 - **v3:** No change.
- [hostPrefixEnabled](#)
 - **v2:** Whether to marshal request parameters to the prefix of hostname.
 - **v3: Deprecated.** SDK *always* injects the hostname prefix when necessary.
- [httpOptions](#)

A set of options to pass to the low-level HTTP request. These options are aggregated differently in v3. You can configure them by supplying a new `requestHandler`. Here's the example of setting http options in Node.js runtime. You can find more in [v3 API reference for NodeHttpHandler](#).

All v3 requests use HTTPS by default. You only need to provide custom `httpsAgent`.

```
const { Agent } = require("https");
const { Agent: HttpAgent } = require("http");
const { NodeHttpHandler } = require("@smithy/node-http-handler");
const dynamodbClient = new DynamoDBClient({
  requestHandler: new NodeHttpHandler({
    httpsAgent: new Agent({
      /*params*/
    }),
    connectionTimeout: /*number in milliseconds*/,
    socketTimeout: /*number in milliseconds*/
  }),
});
```

If you are passing custom endpoint which uses http, then you need to provide `httpAgent`.

```
const { Agent } = require("http");
const { NodeHttpHandler } = require("@smithy/node-http-handler");

const dynamodbClient = new DynamoDBClient({
  requestHandler: new NodeHttpHandler({
    httpAgent: new Agent({
      /*params*/
    }),
  }),
  endpoint: "http://example.com",
});
```

If the client is running in browsers, a different set of options is available. You can find more in [v3 API reference for FetchHttpHandler](#).

```
const { FetchHttpHandler } = require("@smithy/fetch-http-handler");
const dynamodbClient = new DynamoDBClient({
  requestHandler: new FetchHttpHandler({
```

```

    requestTimeout: /* number in milliseconds */
  }},
});

```

Each option of `httpOptions` is specified below:

- `proxy`
 - **v2:** The URL to proxy requests through.
 - **v3:** You can set up a proxy with an agent following [Configuring proxies for Node.js](#).
- `agent`
 - **v2:** The Agent object to perform HTTP requests with. Used for connection pooling.
 - **v3:** You can configure `httpAgent` or `httpsAgent` as shown in the examples above.
- `connectTimeout`
 - **v2:** Sets the socket to timeout after failing to establish a connection with the server after `connectTimeout` milliseconds.
 - **v3:** `connectionTimeout` is available [in NodeHttpHandler options](#).
- `timeout`
 - **v2:** The number of milliseconds a request can take before automatically being terminated.
 - **v3:** `socketTimeout` is available [in NodeHttpHandler options](#).
- `xhrAsync`
 - **v2:** Whether the SDK will send asynchronous HTTP requests.
 - **v3: Deprecated.** Requests are *always* asynchronous.
- `xhrWithCredentials`
 - **v2:** Sets the "withCredentials" property of an XMLHttpRequest object.
 - **v3:** Not available. SDK inherits [the default fetch configurations](#).
- [logger](#)
 - **v2:** An object that responds to `.write()` (like a stream) or `.log()` (like the console object) in order to log information about requests.
 - **v3:** No change. More granular logs are available in v3.
- [maxRedirects](#)
 - **v2:** The maximum amount of redirects to follow for a service request.
 - **v3: Deprecated.** SDK *does not* follow redirects to avoid unintentional cross-region requests.
- [maxRetries](#)
 - **v2:** The maximum amount of retries to perform for a service request.
 - **v3: Changed to maxAttempts.** See more in [v3 API reference for RetryInputConfig](#). Note that `maxAttempts` should be `maxRetries + 1`.

- [paramValidation](#)
 - **v2:** Whether input parameters should be validated against the operation description before sending the request.
 - **v3: Deprecated.** SDK *does not* do validation on client-side at runtime.
- [region](#)
 - **v2:** The region to send service requests to.
 - **v3:** No change. It can also be an async function that returns a region string.
- [retryDelayOptions](#)
 - **v2:** A set of options to configure the retry delay on retryable errors.
 - **v3: Deprecated.** SDK supports more flexible retry strategy with `retryStrategy` client constructor option. See more [in v3 API reference](#).
- [s3BucketEndpoint](#)
 - **v2:** Whether the provided endpoint addresses an individual bucket (false if it addresses the root API endpoint).
 - **v3:** Changed to `bucketEndpoint`. See more in [v3 API reference for bucketEndpoint](#). Note that when set to `true`, you specify the request endpoint in the `Bucket` request parameter, the original endpoint will be overwritten. Whereas in v2, the request endpoint in client constructor overwrites the `Bucket` request parameter.
- [s3DisableBodySigning](#)
 - **v2:** Whether to disable S3 body signing when using signature version v4.
 - **v3:** Renamed to `applyChecksum`.
- [s3ForcePathStyle](#)
 - **v2:** Whether to force path style URLs for S3 objects.
 - **v3:** Renamed to `forcePathStyle`.
- [s3UseArnRegion](#)
 - **v2:** Whether to override the request region with the region inferred from requested resource's ARN.
 - **v3:** Renamed to `useArnRegion`.
- [s3UsEast1RegionalEndpoint](#)
 - **v2:** When region is set to 'us-east-1', whether to send s3 request to global endpoints or 'us-east-1' regional endpoints.
 - **v3: Deprecated.** S3 client will always use regional endpoint if region is set to `us-east-1`. You can set the region to `aws-global` to send requests to S3 global endpoint.
- [signatureCache](#)
 - **v2:** Whether the signature to sign requests with (overriding the API configuration) is cached.

- **v3: Deprecated.** SDK *always* caches the hashed signing keys.
- [signatureVersion](#)
 - **v2:** The signature version to sign requests with (overriding the API configuration).
 - **v3: Deprecated.** Signature V2 supported in v2 SDK has been deprecated by AWS. *v3 only* supports signature v4.
- [sslEnabled](#)
 - **v2:** Whether SSL is enabled for requests.
 - **v3:** Renamed to `tls`.
- [stsRegionalEndpoints](#)
 - **v2:** Whether to send sts request to global endpoints or regional endpoints.
 - **v3: Deprecated.** STS client will *always* use regional endpoints if set to a specific region. You can set the region to `aws-global` to send request to STS global endpoint.
- [useAccelerateEndpoint](#)
 - **v2:** Whether to use the Accelerate endpoint with the S3 service.
 - **v3:** No change.

Credential providers

In v2, the SDK for JavaScript provides a list of credential providers to choose from, as well as a credentials provider chain, available by default on Node.js, that tries to load the AWS credentials from all the most common providers. The SDK for JavaScript v3 simplifies the credential provider's interface, making it easier to use and write custom credential providers. On top of a new credentials provider chain, the SDK for JavaScript v3 all provides a list of credential providers aiming to provide equivalent to v2.

Here are all the credential providers in v2 and their equivalents in v3.

Default Credential Provider

The default credential provider is how the SDK for JavaScript resolve the AWS credential if you *do not* provide one explicitly.

- **v2:** [CredentialProviderChain](#) in Node.js resolves credential from sources as following order:
 - [Environmental variable](#)
 - [Shared credentials file](#)
 - [ECS container credentials](#)

- [Spawning external process](#)
- [OIDC token from specified file](#)
- [EC2 instance metadata](#)

If one of the credential providers above fails to resolve the AWS credential, the chain falls back to next provider until a valid credential is resolved, and the chain will throw an error when all of the providers fail.

In Browser and React Native runtimes, the credential chain is empty, and credentials must be set explicitly.

- **v3:** [defaultProvider](#). The credential sources and fallback order *does not* change in v3. It also supports [AWS IAM Identity Center credentials](#).

Temporary Credentials

- **v2:** [ChainableTemporaryCredentials](#) represents temporary credentials retrieved from `AWS.STS`. Without any extra parameters, credentials will be fetched from the `AWS.STS.getSessionToken()` operation. If an IAM role is provided, the `AWS.STS.assumeRole()` operation will be used to fetch credentials for the role instead. `AWS.ChainableTemporaryCredentials` differs from `AWS.TemporaryCredentials` in the way `masterCredentials` and refreshes are handled. `AWS.ChainableTemporaryCredentials` refreshes expired credentials using the `masterCredentials` passed by the user to support chaining of STS credentials. However, `AWS.TemporaryCredentials` recursively collapses the `masterCredentials` during instantiation, precluding the ability to refresh credentials which require intermediate, temporary credentials.

The original [TemporaryCredentials](#) has been **deprecated** in favor of `ChainableTemporaryCredentials` in v2.

- **v3:** [fromTemporaryCredentials](#). You can call `fromTemporaryCredentials()` from the `@aws-sdk/credential-providers` package. Here's an example:

```
import { FooClient } from "@aws-sdk/client-foo";
import { fromTemporaryCredentials } from "@aws-sdk/credential-providers"; // ES6
import
// const { FooClient } = require("@aws-sdk/client-foo");
// const { fromTemporaryCredentials } = require("@aws-sdk/credential-providers"); //
CommonJS import
```

```
const sourceCredentials = {
  // A credential can be a credential object or an async function that returns a
  // credential object
};
const client = new FooClient({
  credentials: fromTemporaryCredentials({
    masterCredentials: sourceCredentials,
    params: { RoleArn },
  }),
});
```

Amazon Cognito Identity Credentials

Load credentials from the Amazon Cognito Identity service, normally used in browsers.

- **v2:** [CognitoIdentityCredentials](#) Represents credentials retrieved from STS Web Identity Federation using the Amazon Cognito Identity service.
- **v3:** [Cognito Identity Credential Provider](#) The [@aws/credential-providers package](#) provides two credential provider functions, one of which [fromCognitoIdentity](#) takes an identity ID and calls `cognitoIdentity:GetCredentialsForIdentity`, while the other [fromCognitoIdentityPool](#) takes an identity pool ID, calls `cognitoIdentity:GetId` on the first invocation, and then calls `fromCognitoIdentity`. Subsequent invocations of the latter do not re-invoke `GetId`.

The provider implements the "Simplified Flow" described in the [Amazon Cognito Developer Guide](#). The "Classic Flow" which involves calling `cognito:GetOpenIdToken` and then calling `sts:AssumeRoleWithWebIdentity` is *not* supported. Please open a [feature request](#) to us if you need it.

```
// fromCognitoIdentityPool example
import { fromCognitoIdentityPool } from "@aws-sdk/credential-providers"; // ES6
import
// const { fromCognitoIdentityPool } = require("@aws-sdk/credential-providers"); //
// CommonJS import

const client = new FooClient({
  region: "us-east-1",
  credentials: fromCognitoIdentityPool({
    clientConfig: cognitoIdentityClientConfig, // Optional
    identityPoolId: "us-east-1:1699ebc0-7900-4099-b910-2df94f52a030",
```

```

    customRoleArn: "arn:aws:iam::1234567890:role/MYAPP-CognitoIdentity", // Optional
    logins: {
      // Optional
      "graph.facebook.com": "FBTOKEN",
      "www.amazon.com": "AMAZONTOKEN",
      "api.twitter.com": "TWTWITTERTOKEN",
    },
  }),
});

```

```

// fromCognitoIdentity example
import { fromCognitoIdentity } from "@aws-sdk/credential-providers"; // ES6 import
// const { fromCognitoIdentity } = require("@aws-sdk/credential-provider-cognito-identity"); // CommonJS import

const client = new FooClient({
  region: "us-east-1",
  credentials: fromCognitoIdentity({
    clientConfig: cognitoIdentityClientConfig, // Optional
    identityId: "us-east-1:128d0a74-c82f-4553-916d-90053e4a8b0f",
    customRoleArn: "arn:aws:iam::1234567890:role/MYAPP-CognitoIdentity", // Optional
    logins: {
      // Optional
      "graph.facebook.com": "FBTOKEN",
      "www.amazon.com": "AMAZONTOKEN",
      "api.twitter.com": "TWTWITTERTOKEN",
    },
  }),
});

```

EC2 Metadata (IMDS) Credential

Represents credentials received from the metadata service on an Amazon EC2 instance.

- **v2:** [EC2MetadataCredentials](#)
- **v3:** [fromInstanceMetadata](#): Creates a credential provider that will source credentials from the Amazon EC2 Instance Metadata Service.

```

import { fromInstanceMetadata } from "@aws-sdk/credential-providers"; // ES6 import
// const { fromInstanceMetadata } = require("@aws-sdk/credential-providers"); // CommonJS import

```

```
const client = new FooClient({
  credentials: fromInstanceMetadata({
    maxRetries: 3, // Optional
    timeout: 0, // Optional
  }),
});
```

ECS Credentials

Represents credentials received from specified URL. This provider will request temporary credentials from URI specified by the `AWS_CONTAINER_CREDENTIALS_RELATIVE_URI` or the `AWS_CONTAINER_CREDENTIALS_FULL_URI` environment variable.

- **v2:** `ECSCredentials` or [RemoteCredentials](#).
- **v3:** [fromContainerMetadata](#) creates a credential provider that will source credentials from the Amazon ECS Container Metadata Service.

```
import { fromContainerMetadata } from "@aws-sdk/credential-providers"; // ES6 import

const client = new FooClient({
  credentials: fromContainerMetadata({
    maxRetries: 3, // Optional
    timeout: 0, // Optional
  }),
});
```

File System Credentials

- **v2:** [FileSystemCredentials](#) represents credentials from a JSON file on disk.
- **v3: Deprecated.** You can explicitly read the JSON file and supply to the client. Please open a [feature request](#) to us if you need it.

SAML Credential Provider

- **v2:** [SAMLCredentials](#) represents credentials retrieved from STS SAML support.
- **v3: Not available.** Please open a [feature request](#) to us if you need it.

Shared Credential File Credentials

Loads credentials from shared credentials file (defaulting to `~/.aws/credentials` or defined by the `AWS_SHARED_CREDENTIALS_FILE` environment variable). This file is supported across different AWS SDKs and tools. You can refer to the [shared config and credentials files document](#) for more information.

- **v2:** [SharedIniFileCredentials](#)
- **v3:** [fromIni](#).

```
import { fromIni } from "@aws-sdk/credential-providers";
// const { fromIni } from("@aws-sdk/credential-providers");

const client = new FooClient({
  credentials: fromIni({
    configFilepath: "~/.aws/config", // Optional
    filepath: "~/.aws/credentials", // Optional
    mfaCodeProvider: async (mfaSerial) => {
      // implement a pop-up asking for MFA code
      return "some_code";
    }, // Optional
    profile: "default", // Optional
    clientConfig: { region }, // Optional
  }),
});
```

Web Identity Credentials

Retrieves credentials using OIDC token from a file on disk. It's commonly used in EKS.

- **v2:** [TokenFileWebIdentityCredentials](#).
- **v3:** [fromTokenFile](#)

```
import { fromTokenFile } from "@aws-sdk/credential-providers"; // ES6 import
// const { fromTokenFile } from("@aws-sdk/credential-providers"); // CommonJS import

const client = new FooClient({
  credentials: fromTokenFile({
```

```
// Optional. If skipped, read from `AWS_ROLE_ARN` environmental variable
roleArn: "arn:xxxx",
// Optional. If skipped, read from `AWS_ROLE_SESSION_NAME` environmental variable
roleSessionName: "session:a",
// Optional. STS client config to make the assume role request.
clientConfig: { region },
}),
});
```

Web Identity Federation Credentials

Retrieves credentials from STS web identity federation support.

- **v2:** [WebIdentityCredentials](#)
- **v3:** [fromWebToken](#)

```
import { fromWebToken } from "@aws-sdk/credential-providers"; // ES6 import
// const { fromWebToken } from("@aws-sdk/credential-providers"); // CommonJS import

const client = new FooClient({
  credentials: fromWebToken({
    // Optional. If skipped, read from `AWS_ROLE_ARN` environmental variable
    roleArn: "arn:xxxx",
    // Optional. If skipped, read from `AWS_ROLE_SESSION_NAME` environmental variable
    roleSessionName: "session:a",
    // Optional. STS client config to make the assume role request.
    clientConfig: { region },
  }),
});
```

Amazon S3 considerations

Amazon S3 multipart upload

In v2, the Amazon S3 client contains an [upload\(\)](#) operation that supports uploading large objects with [multipart upload feature offered by Amazon S3](#).

In v3, the [@aws-sdk/lib-storage](#) package is available. It supports all the features offered in the v2 `upload()` operation and supports both Node.js and browsers runtime.

Amazon S3 presigned URL

In v2, the Amazon S3 client contains the [getSignedUrl\(\)](#) and [getSignedUrlPromise\(\)](#) operations to generate an URL that users can use to upload or download objects from Amazon S3.

In v3, the [@aws-sdk/s3-request-presigner](#) package is available. This package contains the functions for both [getSignedUrl\(\)](#) and [getSignedUrlPromise\(\)](#) operations. This [blog post](#) discusses the details of this package.

Amazon S3 region redirects

If an incorrect region is passed to the Amazon S3 client and a subsequent `PermanentRedirect` (status 301) error is thrown, the Amazon S3 client in v3 supports region redirects (previously known as the Amazon S3 Global Client in v2). You can use the [followRegionRedirects](#) flag in the client configuration to make the Amazon S3 client follow region redirects and support its function as a global client.

Note

Note that this feature can result in additional latency as failed requests are retried with a corrected region when receiving a `PermanentRedirect` error with status 301. This feature should only be used if you do not know the region of your bucket(s) ahead of time.

Amazon S3 streaming and buffered responses

The v3 SDK prefers not to buffer potentially large responses. This is commonly encountered in the Amazon S3 `GetObject` operation, which returned a `Buffer` in v2, but returns a `Stream` in v3.

For Node.js, you must consume the stream or garbage collect the client or its request handler to keep the connections open to new traffic by freeing sockets.

```
// v2
const get = await s3.getObject({ ... }).promise(); // this buffers consumes the stream
already.
```

```
// v3, consume the stream to free the socket
const get = await s3.getObject({ ... }); // object .Body has unconsumed stream
```

```
const str = await get.Body.transformToString(); // consumes the stream

// other ways to consume the stream include writing it to a file,
// passing it to another consumer like an upload, or buffering to
// a string or byte array.
```

For more information, see section on [socket exhaustion](#).

DynamoDB document client

Basic usage of DynamoDB document client in v3

- In v2, you can use the [AWS.DynamoDB.DocumentClient](#) class to call DynamoDB APIs with native JavaScript types like Array, Number, and Object. It thus simplifies working with items in Amazon DynamoDB by abstracting away the notion of attribute values.
- In v3, the equivalent [@aws-sdk/lib-dynamodb](#) client is available. It's similar to normal service clients from v3 SDK, with the difference that it takes a basic DynamoDB client in its constructor.

Example:

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb"; // ES6 import
// const { DynamoDBClient } = require("@aws-sdk/client-dynamodb"); // CommonJS import
import { DynamoDBDocumentClient, PutCommand } from "@aws-sdk/lib-dynamodb"; // ES6
import
// const { DynamoDBDocumentClient, PutCommand } = require("@aws-sdk/lib-dynamodb"); //
CommonJS import

// Bare-bones DynamoDB Client
const client = new DynamoDBClient({});

// Bare-bones document client
const ddbDocClient = DynamoDBDocumentClient.from(client); // client is DynamoDB client

await ddbDocClient.send(
  new PutCommand({
    TableName,
    Item: {
      id: "1",
      content: "content from DynamoDBDocumentClient",
    },
  },
})
```



```
);
```

Undefined values in when marshalling

- In v2, undefined values in objects were automatically omitted during the marshalling process to DynamoDB.
- In v3, the default marshalling behavior in `@aws-sdk/lib-dynamodb` has changed: objects with undefined values are no longer omitted. To align with v2's functionality, developers must explicitly set the `removeUndefinedValues` to `true` in the `marshallOptions` of the DynamoDB Document Client.

Example:

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, PutCommand } from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});

// The DynamoDBDocumentClient is configured to handle undefined values properly
const ddbDocClient = DynamoDBDocumentClient.from(client, {
  marshallOptions: {
    removeUndefinedValues: true
  }
});

await ddbDocClient.send(
  new PutCommand({
    TableName,
    Item: {
      id: "123",
      content: undefined // This value will be automatically omitted
    }
  })
);
```

More examples and configurations are available in the [package README](#).

Waiters and signers

This page describes the usage of waiters and signers in the AWS SDK for JavaScript v3.

Waiters

In v2, all waiters are bound to the service client class, and you need to specify in waiter's input which designed state the client will be waiting for. For example, you need to call [waitFor\("bucketExists"\)](#) to wait for a newly created bucket to be ready.

In v3, you don't need to import waiters if your application doesn't need one. Moreover, you can import only the waiter you need to wait for the particular desired state you want. Thus, you can reduce your bundle size and improve performance. Here is an example of waiting for bucket to be ready after creation:

```
import { S3Client, CreateBucketCommand, waitUntilBucketExists } from "@aws-sdk/client-s3"; // ES6 import
// const { S3Client, CreateBucketCommand, waitUntilBucketExists } = require("@aws-sdk/client-s3"); // CommonJS import

const Bucket = "BUCKET_NAME";
const client = new S3Client({ region: "REGION" });
const command = new CreateBucketCommand({ Bucket });

await client.send(command);
await waitUntilBucketExists({ client, maxWaitTime: 60 }, { Bucket });
```

You can find everything of how to configure the waiters in the [blog post of waiters in the AWS SDK for JavaScript v3](#).

Amazon CloudFront Signer

In v2, you can sign the request to access restricted Amazon CloudFront distributions with [AWS.CloudFront.Signer](#).

In v3, you have the same utilities provided in the [@aws-sdk/cloudfront-signer](#) package.

Amazon RDS Signer

In v2, you can generate the auth token to an Amazon RDS database using [AWS.RDS.Signer](#).

In v3, the similar utility class is available in [@aws-sdk/rds-signer](#) package.

Amazon Polly Signer

In v2, you can generate a signed URL to the speech synthesized by Amazon Polly service with [AWS.Polly.Presigner](#).

In v3, the similar utility function is available in [@aws-sdk/polly-request-presigner](#) package.

Notes on specific service clients

AWS Lambda

Lambda invocations response type differs in v2 and v3.

```
// v2
import { Lambda } from "@aws-sdk/client-lambda";
import AWS from "aws-sdk";

const lambda = new AWS.Lambda({ REGION });
const invoke = await lambda.invoke({
  FunctionName: "echo",
  Payload: JSON.stringify({ message: "hello" }),
}).promise();

// in v2, Lambda::invoke::Payload is automatically converted to string via a
// specific code customization.
const payloadIsString = typeof invoke.Payload === "string";
console.log("Invoke response payload type is string:", payloadIsString);

const payloadObject = JSON.parse(invoke.Payload);
console.log("Invoke response object", payloadObject);
```

```
// v3
const lambda = new Lambda({ REGION });
const invoke = await lambda.invoke({
  FunctionName: "echo",
  Payload: JSON.stringify({ message: "hello" }),
});

// in v3, Lambda::invoke::Payload is not automatically converted to a string.
// This is to reduce the number of customizations that create inconsistent behaviors.
const payloadIsByteArray = invoke.Payload instanceof Uint8Array;
```

```
console.log("Invoke response payload type is Uint8Array:", payloadIsByteArray);

// To maintain the old functionality, only one additional method call is needed:
// v3 adds a method to the Uint8Array called transformToString.
const payloadObject = JSON.parse(invoke.Payload.transformToString());
console.log("Invoke response object", payloadObject);
```

Amazon SQS

MD5 Checksum

To skip computation of MD5 checksums of message bodies, set `md5` to *false* on the configuration object. Otherwise, the SDK by default will calculate the checksum for sending messages, as well as validating the checksum for retrieved messages.

```
// Example: Skip MD5 checksum in Amazon SQS
import { SQS } from "@aws-sdk/client-sqs";

new SQS({
  md5: false // note: only available in v3.547.0 and higher
});
```

When using a custom `QueueUrl` in Amazon SQS operations that have this as an input parameter, in v2 it was possible to supply a custom `QueueUrl` which would override the Amazon SQS Client's default endpoint.

Multi-region messages

You should use one client per region in v3. The AWS region is meant to be initialized at the client level and not changed between requests.

```
import { SQS } from "@aws-sdk/client-sqs";

const sqsClients = {
  "us-east-1": new SQS({ region: "us-east-1" }),
  "us-west-2": new SQS({ region: "us-west-2" }),
};

const queues = [
  { region: "us-east-1", url: "https://sqs.us-east-1.amazonaws.com/{AWS_ACCOUNT}/MyQueue" },
```

```
{ region: "us-west-2", url: "https://sqs.us-west-2.amazonaws.com/{AWS_ACCOUNT}/MyOtherQueue" },
];

for (const { region, url } of queues) {
  const params = {
    MessageBody: "Hello",
    QueueUrl: url,
  };
  await sqsClients[region].sendMessage(params);
}
```

Custom endpoint

In v3, when using a custom endpoint, i.e. one that differs from the default public Amazon SQS endpoints, you should always set the endpoint on the Amazon SQS Client as well as the `QueueUrl` field.

```
import { SQS } from "@aws-sdk/client-sqs";

const sqs = new SQS({
  // client endpoint should be specified in v3 when not the default public SQS endpoint
  // for your region.
  // This is required for versions <= v3.506.0
  // This is optional but recommended for versions >= v3.507.0 (a warning will be
  // emitted)
  endpoint: "https://my-custom-endpoint:8000/",
});

await sqs.sendMessage({
  QueueUrl: "https://my-custom-endpoint:8000/1234567/MyQueue",
  Message: "hello",
});
```

If you are not using a custom endpoint, then you do not need to set endpoint on the client.

```
import { SQS } from "@aws-sdk/client-sqs";

const sqs = new SQS({
  region: "us-west-2",
});
```

```
await sqs.sendMessage({
  QueueUrl: "https://sqs.us-west-2.amazonaws.com/1234567/MyQueue",
  Message: "hello",
});
```

Document history for AWS SDK for JavaScript version 3

Document History

The following table describes the important changes in the V3 release of the *AWS SDK for JavaScript* from October 20, 2020, onward. For notification about updates to this documentation, you can subscribe to an [RSS feed](#).

Change	Description	Date
Announcement	Updated top banner with an end-of-support reminder for Internet Explorer 11.	September 23, 2022
Minor updates	Minor updates to clarity and resolving broken links. Added awareness links to AWS SDKs and Tools Reference Guide.	August 22, 2022
Enforce a minimum TLS version	Added information about TLS 1.3.	March 31, 2022
Updated AWS Lambda tutorial	Added tutorial demonstrating how to build a browser-based application for submitting data to a Amazon DynamoDB table.	October 20, 2020
Set credentials in Node.js topic updated	Update topic about setting credentials in Node.js for AWS SDK for JavaScript V3.	October 20, 2020
Migrate to v3	Added topic to describe how to migrate to AWS SDK for JavaScript v3.	October 20, 2020
Getting Started	Updated topics for getting started in the browser and	October 20, 2020

	getting started with Node.js for AWS SDK for JavaScript V3.	
Browser builder	Information about AWS Browser Builder was removed because it is not required for AWS SDK for JavaScript V3.	October 20, 2020
Amazon Transcribe service examples updated	Updated Amazon Transcribe service examples for AWS SDK for JavaScript V3.	October 20, 2020
Amazon Simple Notification Service service examples updated	Updated Amazon Simple Notification Service service examples for AWS SDK for JavaScript V3.	October 20, 2020
Amazon Simple Email Service service examples updated	Updated Amazon Simple Email Service service examples for AWS SDK for JavaScript V3.	October 20, 2020
Amazon Redshift service examples updated	Updated Amazon Redshift service examples for AWS SDK for JavaScript V3.	October 20, 2020
Amazon Lex service examples updated	Updated Amazon Lex service examples for AWS SDK for JavaScript V3.	October 20, 2020
Amazon DynamoDB service examples updated	Updated Amazon DynamoDB service examples for AWS SDK for JavaScript V3.	October 20, 2020
AWS Elemental MediaConvert service examples updated	Updated AWS Elemental MediaConvert service examples for AWS SDK for JavaScript V3.	October 20, 2020

[AWS Lambda service examples updated](#)

Updated AWS Lambda service examples for AWS SDK for JavaScript V3. October 20, 2020

[AWS SDK for JavaScript V3 Developer Guide preview](#)

Released pre-release version of the AWS SDK for JavaScript V3 Developer Guide. October 19, 2020