# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>What's the AWS SDK for JavaScript?</td>
<td>1</td>
</tr>
<tr>
<td>Maintenance and support for SDK major versions</td>
<td>1</td>
</tr>
<tr>
<td>What's new in Version 3</td>
<td>2</td>
</tr>
<tr>
<td>Modularized packages</td>
<td>2</td>
</tr>
<tr>
<td>New middleware stack</td>
<td>7</td>
</tr>
<tr>
<td>Using the SDK with Node.js</td>
<td>7</td>
</tr>
<tr>
<td>Using the SDK with AWS Cloud9</td>
<td>8</td>
</tr>
<tr>
<td>Using the SDK with AWS Amplify</td>
<td>8</td>
</tr>
<tr>
<td>Using the SDK with web browsers</td>
<td>8</td>
</tr>
<tr>
<td>Using browsers in V3</td>
<td>9</td>
</tr>
<tr>
<td>Common use cases</td>
<td>9</td>
</tr>
<tr>
<td>About the examples</td>
<td>10</td>
</tr>
<tr>
<td>Resources</td>
<td></td>
</tr>
<tr>
<td>Get started</td>
<td>11</td>
</tr>
<tr>
<td>SDK authentication with AWS</td>
<td>11</td>
</tr>
<tr>
<td>Start an AWS access portal session</td>
<td>12</td>
</tr>
<tr>
<td>More authentication information</td>
<td>13</td>
</tr>
<tr>
<td>Get started with Node.js</td>
<td>14</td>
</tr>
<tr>
<td>The scenario</td>
<td>14</td>
</tr>
<tr>
<td>Prerequisites</td>
<td>14</td>
</tr>
<tr>
<td>Step 1: Set up the package structure and installing client packages</td>
<td>14</td>
</tr>
<tr>
<td>Step 2: Add necessary imports and SDK code</td>
<td>15</td>
</tr>
<tr>
<td>Step 3: Run the example</td>
<td>18</td>
</tr>
<tr>
<td>Get started in the browser</td>
<td>18</td>
</tr>
<tr>
<td>The Scenario</td>
<td>19</td>
</tr>
<tr>
<td>Step 1: Create an Amazon Cognito identity pool and IAM role</td>
<td>19</td>
</tr>
<tr>
<td>Step 2: Add a policy to the created IAM role</td>
<td>20</td>
</tr>
<tr>
<td>Step 3: Add a Amazon S3 bucket and object</td>
<td>21</td>
</tr>
<tr>
<td>Step 4: Set up the browser code</td>
<td>22</td>
</tr>
<tr>
<td>Step 5: Run the Example</td>
<td>23</td>
</tr>
<tr>
<td>Cleanup</td>
<td>23</td>
</tr>
<tr>
<td>Setting up the SDK for JavaScript</td>
<td>24</td>
</tr>
<tr>
<td>Prerequisites</td>
<td>24</td>
</tr>
<tr>
<td>Using a callback function</td>
<td>58</td>
</tr>
<tr>
<td>---------------------------</td>
<td>----</td>
</tr>
<tr>
<td>Creating service client requests</td>
<td>59</td>
</tr>
<tr>
<td>Handling service client responses</td>
<td>61</td>
</tr>
<tr>
<td>Accessing data returned in the response</td>
<td>61</td>
</tr>
<tr>
<td>Accessing error information</td>
<td>61</td>
</tr>
<tr>
<td>Working with JSON</td>
<td>61</td>
</tr>
<tr>
<td>JSON as service object parameters</td>
<td>62</td>
</tr>
<tr>
<td>Code examples subset with guidance</td>
<td>63</td>
</tr>
<tr>
<td>JavaScript ES6/CommonJS syntax</td>
<td>64</td>
</tr>
<tr>
<td>Amazon DynamoDB examples</td>
<td>67</td>
</tr>
<tr>
<td>AWS Elemental MediaConvert examples</td>
<td>91</td>
</tr>
<tr>
<td>AWS Lambda examples</td>
<td>113</td>
</tr>
<tr>
<td>Amazon Lex examples</td>
<td>113</td>
</tr>
<tr>
<td>Amazon Polly examples</td>
<td>114</td>
</tr>
<tr>
<td>Amazon Redshift examples</td>
<td>117</td>
</tr>
<tr>
<td>Amazon SES examples</td>
<td>125</td>
</tr>
<tr>
<td>Amazon SNS Examples</td>
<td>153</td>
</tr>
<tr>
<td>Amazon Transcribe examples</td>
<td>187</td>
</tr>
<tr>
<td>Cross-service: Setting up Node.js on an Amazon EC2 instance</td>
<td>198</td>
</tr>
<tr>
<td>Cross-service: App to submit data</td>
<td>201</td>
</tr>
<tr>
<td>Cross-service: Transcription app</td>
<td>209</td>
</tr>
<tr>
<td>Cross-service: Amazon API Gateway and Lambda</td>
<td>221</td>
</tr>
<tr>
<td>Cross-service: Serverless workflows with Step Functions</td>
<td>236</td>
</tr>
<tr>
<td>Cross-service: Scheduled Lambda events</td>
<td>252</td>
</tr>
<tr>
<td>Cross-service: Amazon Lex example</td>
<td>264</td>
</tr>
<tr>
<td>Cross-service: Messaging app</td>
<td>278</td>
</tr>
</tbody>
</table>

**Using AWS Cloud9 with the SDK for JavaScript** ................................................................. 292

| Step 1: Set up your AWS account to use AWS Cloud9 | 292 |
| Step 2: Set up your AWS Cloud9 development environment | 292 |
| Step 3: Set up the SDK for JavaScript | 293 |
| To set up the SDK for JavaScript for Node.js | 293 |
| To set up the SDK for JavaScript in the browser | 294 |
| Step 4: Download example code | 294 |
| Step 5: Run and debug example code | 294 |

**Code examples** ........................................................................................................ 296

<p>| Actions and scenarios | 296 |</p>
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto Scaling</td>
<td>297</td>
</tr>
<tr>
<td>CloudWatch</td>
<td>340</td>
</tr>
<tr>
<td>CloudWatch Events</td>
<td>357</td>
</tr>
<tr>
<td>CloudWatch Logs</td>
<td>364</td>
</tr>
<tr>
<td>CodeBuild</td>
<td>374</td>
</tr>
<tr>
<td>Amazon Cognito Identity Provider</td>
<td>377</td>
</tr>
<tr>
<td>DynamoDB</td>
<td>397</td>
</tr>
<tr>
<td>Amazon EC2</td>
<td>444</td>
</tr>
<tr>
<td>Elastic Load Balancing</td>
<td>526</td>
</tr>
<tr>
<td>EventBridge</td>
<td>576</td>
</tr>
<tr>
<td>AWS Glue</td>
<td>583</td>
</tr>
<tr>
<td>Health Imaging</td>
<td>608</td>
</tr>
<tr>
<td>IAM</td>
<td>644</td>
</tr>
<tr>
<td>Lambda</td>
<td>754</td>
</tr>
<tr>
<td>Amazon Personalize</td>
<td>764</td>
</tr>
<tr>
<td>Amazon Personalize Events</td>
<td>781</td>
</tr>
<tr>
<td>Amazon Personalize Runtime</td>
<td>785</td>
</tr>
<tr>
<td>Amazon Pinpoint</td>
<td>789</td>
</tr>
<tr>
<td>Amazon Redshift</td>
<td>799</td>
</tr>
<tr>
<td>Amazon S3</td>
<td>804</td>
</tr>
<tr>
<td>S3 Glacier</td>
<td>843</td>
</tr>
<tr>
<td>SageMaker</td>
<td>847</td>
</tr>
<tr>
<td>Secrets Manager</td>
<td>880</td>
</tr>
<tr>
<td>Amazon SES</td>
<td>882</td>
</tr>
<tr>
<td>Amazon SNS</td>
<td>905</td>
</tr>
<tr>
<td>Amazon SQS</td>
<td>944</td>
</tr>
<tr>
<td>AWS STS</td>
<td>981</td>
</tr>
<tr>
<td>AWS Support</td>
<td>984</td>
</tr>
<tr>
<td>Amazon Transcribe</td>
<td>1002</td>
</tr>
<tr>
<td>Cross-service examples</td>
<td>1010</td>
</tr>
<tr>
<td>Build an Amazon Transcribe app</td>
<td>1011</td>
</tr>
<tr>
<td>Build an Amazon Transcribe streaming app</td>
<td>1012</td>
</tr>
<tr>
<td>Build an app to submit data to a DynamoDB table</td>
<td>1012</td>
</tr>
<tr>
<td>Building an Amazon Lex chatbot</td>
<td>1013</td>
</tr>
<tr>
<td>Create a serverless application to manage photos</td>
<td>1013</td>
</tr>
<tr>
<td>Create a web application to track DynamoDB data</td>
<td>1014</td>
</tr>
</tbody>
</table>
Create an Aurora Serverless work item tracker ................................................................. 1014
Create an Amazon Textract explorer application .............................................................. 1015
Create an application to analyze customer feedback ......................................................... 1015
Detect PPE in images ........................................................................................................ 1019
Detect objects in images .................................................................................................... 1020
Detect people and objects in a video .................................................................................. 1021
Invoke a Lambda function from a browser ....................................................................... 1021
Use API Gateway to invoke a Lambda function ................................................................. 1022
Use Step Functions to invoke Lambda functions ............................................................... 1023
Use scheduled events to invoke a Lambda function ........................................................... 1023

**Security** .......................................................................................................................... 1025
Data protection .................................................................................................................. 1025
Identity and Access Management ...................................................................................... 1026
  Audience .......................................................................................................................... 1027
  Authenticating with identities ......................................................................................... 1027
  Managing access using policies ...................................................................................... 1031
  How AWS services work with IAM .............................................................................. 1033
    Troubleshooting AWS identity and access ................................................................ 1033
Compliance Validation ........................................................................................................ 1035
Resilience ............................................................................................................................ 1036
Infrastructure Security ....................................................................................................... 1037
  Enforcing a minimum TLS version ............................................................................... 1038
    Verify and enforce TLS in Node.js ............................................................................ 1038
    Verify and enforce TLS in a browser script .............................................................. 1040

**Document history** ......................................................................................................... 1043
Document History ............................................................................................................. 1043
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.

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
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.

```javascript
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).

```javascript
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.

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

(async function () {
    const dbclient = new DynamoDBClient({ region: 'us-west-2'});
    try {
        const results = await dbclient.send(new ListTablesCommand);
        results.TableNames.forEach(function (item, index) {
            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.

```javascript
// 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-\textit{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.

```javascript
import { DynamoDB, CreateTableCommand } from '@aws-sdk/client-dynamodb';
```

To call these commands in the recommended async/await pattern, use the following syntax.

```javascript
CLIENT.send(new XXXCommand)
```

For example, the following example creates a DynamoDB table using the recommended async/await pattern.

```javascript
import { DynamoDB, CreateTableCommand } from '@aws-sdk/client-dynamodb';
const dynamodb = new DynamoDB({region: 'us-west-2'});
var tableParams = {
    Table : \textit{TABLE\_NAME}
};
(async function () => {
    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.

```javascript
const { DynamoDB } = require('@aws-sdk/client-dynamodb');
```
To call V2 commands in the recommended async/await pattern, use the following syntax.

\[
\text{client.command(parameters)}
\]

The following example uses the V2 createTable command to create a DynamoDB table using the recommended async/await pattern.

```javascript
const {DynamoDB} = require('@aws-sdk/client-dynamodb');
const dynamoDB = new DynamoDB({region: 'us-west-2'});
var tableParams = {
    TableName : \text{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.

```javascript
const {S3} = require('@aws-sdk/client-s3');
const s3 = new S3({region: 'us-west-2'});
var bucketParams = {
    Bucket : \text{BUCKET\_NAME}
};
function run(){
    s3.createBucket(bucketParams, function(err, data) {
        if (err) {
            console.log("Error", err);
        } else {
            console.log("Success", data.Location);
        }
    });
};
```
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.

```javascript
dbclient.middlewareStack.add(
  (next, context) => args => {
    args.request.headers['Custom-Header'] = 'value';
    return next(args);
  },
  {
    step: 'build'
  }
);

dbclient.send(new PutObjectCommand(params));
```

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 Using 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 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:

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

Note
For more information about Webpack, see Bundling 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 Using 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 AWS config 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.

```ini
[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 SS0OIDC) 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 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:
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:

```json
{
    "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.

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

import {
    S3Client,
    PutObjectCommand,
    CreateBucketCommand,

```
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({

```javascript
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/](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](https://www.amazon.com/iam/)

   **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:

   ```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](https://github.com/awsdocs/aws-doc-sdk-examples).

**To set up the example application**

1. **Install** [Node.js](https://nodejs.org).
2. From the command line, clone the [AWS Code Examples Repository](https://github.com/awsdocs/aws-doc-sdk-examples):

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

3. Navigate to the example application:

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

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

   ```bash
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:

   \texttt{cd aws-doc-sdk-examples/javascriptv3/example_code/web/s3/list-objects/}

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

   \texttt{npm run dev}

   \texttt{VITE v4.3.9 ready in 280 ms}

   \texttt{# Local: http://localhost:5173/}
   \texttt{# Network: use --host to expose}
   \texttt{# 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, \texttt{getstartedbucket}).
- In the IAM console, delete the role name (for example, \texttt{getStartedRole}).
- In the Amazon Cognito console, delete the identity pool name (for example, \texttt{getStartedPool}).
Setting 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
- Installing the SDK for JavaScript
- Loading the SDK for JavaScript
- Migrating your code to SDK for JavaScript V3

Prerequisites

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

Topics

- Setting up an AWS Node.js environment
- Supported web browsers

Setting 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.

<table>
<thead>
<tr>
<th>Browser</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google Chrome</td>
<td>80.0+</td>
</tr>
<tr>
<td>Mozilla Firefox</td>
<td>80.0+</td>
</tr>
<tr>
<td>Opera</td>
<td>63.0+</td>
</tr>
<tr>
<td>Microsoft Edge</td>
<td>80.0+</td>
</tr>
<tr>
<td>Apple Safari</td>
<td>14.1+</td>
</tr>
<tr>
<td>Samsung Internet</td>
<td>12.0+</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Browser</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google Chrome</td>
<td>49.0+</td>
</tr>
<tr>
<td>Mozilla Firefox</td>
<td>45.0+</td>
</tr>
<tr>
<td>Opera</td>
<td>36.0+</td>
</tr>
<tr>
<td>Microsoft Edge</td>
<td>12.0+</td>
</tr>
<tr>
<td>Windows Internet Explorer</td>
<td>N/A</td>
</tr>
<tr>
<td>Apple Safari</td>
<td>9.0+</td>
</tr>
<tr>
<td>Android Browser</td>
<td>76.0+</td>
</tr>
<tr>
<td>UC Browser</td>
<td>12.12+</td>
</tr>
<tr>
<td>Samsung Internet</td>
<td>5.0+</td>
</tr>
</tbody>
</table>

### Note
Frameworks such as AWS Amplify might not offer the same browser support as the SDK for JavaScript. See the [AWS Amplify Documentation](https://aws-amplify.github.io/amplify-android-documentation/) for details.

## Installing 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](https://www.npmjs.com/), enter the following command at the command prompt, where `SERVICE` is the name of a service, such as `s3`.

```bash
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](https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/).

---

### Installing the SDK

To install the AWS SDK for JavaScript, run the following command in your terminal:

```bash
npm install aws-sdk
```

For a full list of the AWS SDK for JavaScript service client packages, see the [AWS SDK for JavaScript API Reference guide](https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/).
Loading 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, use the following.

```javascript
import {S3} from "@aws-sdk/client-s3";
```

Migrating your code to SDK for JavaScript V3

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 SDK for JavaScript v3, we recommend using the codemod scripts described below.

Using codemod

The experimental collection of codemod scripts in [aws-sdk-js-codemod](https://github.com/aws/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.

```javascript
// example.ts
import AWS from "aws-sdk";

const region = "us-west-2";
const client = new AWS.DynamoDB({ region });
client.listTables({}, (err, data) => {
  if (err) console.log(err, err.stack);
  else console.log(data);
});
```

You can run our `v2-to-v3` transform on `example.ts` as follows.
The transform will convert the DynamoDB import to v3, create v3 client and call the `listTables` operation as follows.

```typescript
// example.ts
import { DynamoDB } from '@aws-sdk/client-dynamodb';

const region = "us-west-2";
const client = new DynamoDB({ region });
client.listTables({}, (err, data) => {
  if (err) console.log(err, err.stack);
  else console.log(data);
});
```

We've implemented transforms for common use cases. If your code doesn't transform correctly, please create a [bug report](https://github.com/aws/aws-sdk-js-v3/issues/new/choose) or [feature request](https://github.com/aws/aws-sdk-js-v3/issues/new/choose) with example input code and observed/expected output code. If your specific use case is already reported in [an existing issue](https://github.com/aws/aws-sdk-js-v3/issues), show your support by an upvote.
Configuring 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
- Setting the AWS Region
- Setting 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.

Setting 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 Regions, create an Amazon EC2 service object for each Region and then set the Region configuration of each service object accordingly.

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

Setting 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.

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

Using 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.

Using 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](https://aws.amazon.com/documentation/sdk-for-javascript/guides/configuration/) 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.

## Setting 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

- [Setting credentials in Node.js](#)
- [Setting credentials in a web browser](#)
Setting 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.
| **AWS SDK for JavaScript**  
**API Reference credential provider method by precedence** | **Credential provider(s) available** | **AWS SDKs and Tools Reference Guide** |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fromEnv()</code></td>
<td>AWS access keys from environment variables</td>
<td></td>
</tr>
<tr>
<td><code>fromSSO()</code></td>
<td>AWS IAM Identity Center. In this guide, see <a href="https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/security-mechanisms.html">SDK authentication with AWS</a>.</td>
<td>IAM Identity Center credential provider</td>
</tr>
<tr>
<td><code>fromIni()</code></td>
<td>AWS access keys from shared config and credentials files</td>
<td>AWS access keys</td>
</tr>
<tr>
<td></td>
<td>Trusted entity provider (such as <code>AWS_ROLE_ARN</code>)</td>
<td>Assume an IAM role</td>
</tr>
<tr>
<td></td>
<td>Web identity token from AWS Security Token Service (AWS STS)</td>
<td>Federate with web identity or OpenID Connect</td>
</tr>
<tr>
<td></td>
<td>Amazon Elastic Container Service (Amazon ECS) credentials</td>
<td>Container credential provider</td>
</tr>
<tr>
<td></td>
<td>Amazon Elastic Compute Cloud (Amazon EC2) instance profile credentials (IMDS credential provider)</td>
<td>IMDS credential provider</td>
</tr>
<tr>
<td></td>
<td>Process credential provider</td>
<td>Process credential provider</td>
</tr>
<tr>
<td><code>fromProcess()</code></td>
<td>Process credential provider</td>
<td>Process credential provider</td>
</tr>
</tbody>
</table>
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**
- Loading credentials in Node.js from IAM roles for Amazon EC2
- Loading credentials for a Node.js Lambda function

**Loading 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.

**Loading 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.

**Setting 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**

- Using Amazon Cognito Identity to authenticate users
Using 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.

Configuring 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.

```javascript
// 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
    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
    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;
};
Switching 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:

```javascript
// Import the required AWS SDK for JavaScript v3 modules.
import {fromCognitoIdentityPool} from '@aws-sdk/credential-providers';
// Set the default credentials.
const creds = new 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.

```javascript
// 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.

Using 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.

```javascript
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.

```javascript
// 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.

Using 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.

**Configuring 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.

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

var dynamodbClient = new DynamoDBClient({
  requestHandler: new NodeHttpHandler({
    httpsAgent: agent
  })
});
```

When using the default of https, the SDK takes the `maxSockets` value from the `globalAgent`. If the `maxSockets` value is not defined, the SDK assumes a `maxSockets` value of 50.

For more information about setting `maxSockets` in Node.js, see the [Node.js online documentation](https://nodejs.org/en/docs/guide/max-sockets/).

**Reusing 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 SDK for JavaScript reuses TCP connections.

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.
To disable reusing TCP connections, set the `AWS_NODEJS_CONNECTION_REUSE_ENABLED` environment variable to false (the default is true).

You can also disable keeping these connections alive on a per-service client basis, as shown in the following example for a DynamoDB client.

```javascript
import { DynamoDBClient } from '@aws-sdk/client-dynamodb';
import { NodeHttpHandler } from '@smithy/node-http-handler';
import { Agent } from 'http';
const dynamodbClient = new DynamoDBClient({
  requestHandler: new NodeHttpHandler({
    httpAgent: 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.

## Configuring 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.

```javascript
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
  })
});
```
### Registering 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.

```javascript
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
Building 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 your require, into a single Javascript file, and load it into browser scripts using a `<script>` tag. For more information about Webpack, see Bundling 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.

Using 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 Bundling 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.

Configuring 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?](https://docs.aws.amazon.com/AmazonS3/latest/dev/CORS-Quick-Ref.html) 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
<?xml version="1.0" encoding="UTF-8"?>
<CORSConfiguration xmlns="http://s3.amazonaws.com/doc/2006-03-01/">
  <CORSRule>
    <AllowedOrigin>https://example.org</AllowedOrigin>
    <AllowedMethod>HEAD</AllowedMethod>
    <AllowedMethod>GET</AllowedMethod>
    <AllowedMethod>PUT</AllowedMethod>
    <AllowedMethod>POST</AllowedMethod>
    <AllowedMethod>DELETE</AllowedMethod>
    <AllowedHeader>*</AllowedHeader>
    <ExposeHeader>ETag</ExposeHeader>
    <ExposeHeader>x-amz-meta-custom-header</ExposeHeader>
  </CORSRule>
</CORSConfiguration>
```

### JSON

```json
[  
  {
    "AllowedHeaders": ["*"],
    "AllowedMethods": ["HEAD", "GET",
```
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.

**Bundling 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](https://github.com/webpack) on GitHub.
Installing 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
```

Configuring 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.

```
// 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.

Running 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.

```json
{
  "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.

```bash
npm run build
```

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

**Using 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.

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

**Bundling for Node.js**

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

```javascript
// Import path for resolving file paths
var path = require("path");
module.exports = {
```

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.

```javascript
// 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"]}
 */
};
Working with 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

- Creating and calling service objects
- Calling services asynchronously
- Creating service client requests
- Handling service client responses
- Working with JSON
- SDK for JavaScript code examples

Creating and calling 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.

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

### Specifying 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 `GetObject` method. For more information about passing JSON parameters, see Working with JSON.

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

You can also call the `GetObjectCommand` method from the `S3Client`:

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

For more information about Amazon S3 parameters, see Class: `S3` in the API Reference.

### Calling 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

• Managing asynchronous calls
• Using async/await
• Using JavaScript promises
• Using an anonymous callback function

Managing 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.

Using 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 Setting credentials.

```javascript
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('
'));
    } catch (err) {
        console.error(err)
    }
})();
```

**Note**

Not all browsers support async/await. See Async functions for a list of browsers with async/await support.

### Using 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.

```javascript
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('
'));
    })
```
Coordinating 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.

```javascript
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.

Using 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.

```javascript
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.

```javascript
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.

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

### Creating 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.

```javascript
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('
'));
  } catch (err) {
    console.error(err);
  }
})();
```
Handling 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.

Accessing data returned in the response

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

In Creating service client requests, the `ListTablesCommand` command returned the table names in the `TableNames` property of the response.

Accessing error information

If a command fails, it throws an exception. You can handle the exception as you need.

Working 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](http://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.

```javascript
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.

```javascript
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.

```javascript
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
• Build a transcription app with authenticated users
• Invoking Lambda with API Gateway
• Creating AWS serverless workflows using AWS SDK for JavaScript
• 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:

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

To its CommonJS equivalent:

```javascript
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:

```javascript
export {s3}
```

To its CommonJS equivalent:

```javascript
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`

```javascript
// 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`

```javascript
// 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" };

JavaScript ES6/CommonJS syntax
// 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
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.

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.

```javascript
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.

```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);
  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](https://github.com).

**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.

```javascript
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](https://github.com).

**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.

```javascript
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.

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

This example code can be found [here on GitHub](https://github.com).

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:
AWS SDK for JavaScript

Developer Guide for SDK Version 3

- 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.

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

const client = new DynamoDBClient({});

export const main = async () => {
  const command = new PutItemCommand({
    TableName: "Cookies",
    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](https://github.com/awsdocs/aws-sdk-js-docs/tree/master/examples/dynamodb/put-item.js).

**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';

code: 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.

```javascript
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.

```javascript
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](https://github.com/aws-sdk-js/aws-sdk/tree/masterdocsExamples/node).

**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.

```javascript
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"]);
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](https://github.com).
```javascript
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;

```

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](https://github.com/aws/aws-sdk-js-v3).

### 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.
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.

```javascript
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}
`);  

  });
};
```

Amazon DynamoDB examples
To run the example, enter the following at the command prompt.

code snippet
node query.js

text
This example code can be found [here on GitHub](https://github.com).

**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.

code snippet
```javascript
import { DynamoDBClient, ScanCommand } from '@aws-sdk/client-dynamodb';

const client = new DynamoDBClient({});

export const main = async () => {
    const command = new ScanCommand({
        TableName: 'Pies',
        ProjectionExpression: 'Flavor, CrustType, Description',
        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' },
        },
    });

    const response = await client.send(command);
    response.Items.forEach(function (pie) {
        console.log(`${pie.Flavor.S} - ${pie.Description.S}
`);
    });
};
```
To run the example, enter the following at the command prompt.

```javascript
node scan.js
```

This example code can be found [here on GitHub](https://github.com).

**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](https://github.com) on GitHub. For more information about programming with Amazon DynamoDB, see [Programming with DynamoDB](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/) 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:

- [DeleteCommand](https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/dynamodb-delete.html)
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 to 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.

```javascript
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](https://github.com).

**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](https://github.com).

**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](https://docs.aws.amazon.com/sdk-for-js/v3/api/module-put.html) that define the attributes of the item to update with tokens you assign values to in the `ExpressionAttributeValues` parameters. Call the `UpdateCommand` method of the DynamoDB Document Client.

```javascript
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](https://github.com).
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](https://github.com).

**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.

```javascript
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](https://github.com/).  

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.

```javascript
// 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.

AWS Elemental MediaConvert examples
This example code can be found [here on GitHub](https://github.com).

**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](https://github.com).
• 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).

```javascript
// 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.

```javascript
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](https://github.com).

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](https://console.aws.amazon.com/mediaconvert) 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".

```json
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",
}
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.

```javascript
const run = async () => {
  try {
    // JSON parameters
    
    // Call run method
    
  } catch (err) {
  }
};
```
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 };
Note

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

---

```javascript
// 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.

```bash
node ec2_canceljob.js
```

This example code can be found [here on GitHub](https://github.com/awsdocs/aws-sdk-js-sdk-v3/tree/master/docs/examples/mediacodec/)

### 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.

```javascript
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](https://github.com).

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.

```javascript
// 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](https://github.com).

### 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.

```javascript
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.

```
// 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: "PROGRESSIVEDOWNLOAD",
  },
  NameModifier: "_1",
},
],
AdAvailOffset: 0,
Inputs: [
  {
    AudioSelectors: {
      "Audio Selector 1": {
        Offset: 0,
        DefaultSelection: "NOT_DEFAULT",
        ProgramSelection: 1,
        SelectorType: "TRACK",
        Tracks: [1],
      },
    },
    VideoSelector: {

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](https://github.com/awsdocs/aws-sdk-for-javascript-v3/tree/master/code_samples/sample-code/mediaconvert).

**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](https://github.com/aws-sdk-docs-examples).

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".

```javascript
// 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: "JOB_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,
                    }
                }
            }
        ]
    }
};
```
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 };
To run the example, enter the following at the command prompt.

```
node emc_listtemplates.js
```

This example code can be found [here on GitHub](https://github.com/awslabs/aws-sdk-js-examples/tree/master/mediamath/examples).

**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.

```javascript
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](https://github.com/awslabs/aws-sdk-js-examples/tree/master/mediamath/examples).

Create a Node.js module with the file name `emc_deletemediatemplate.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.

```javascript
// 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 {
```

AWS Elemental MediaConvert examples
```
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

```json
{
}
```
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.

```javascript
const { StartSpeechSynthesisTaskCommand } = require('@aws-sdk/client-polly');
const { pollyClient } = require('./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](https://github.com/aws-samples/speech-synthesis).

### 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.

```javascript
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.

```javascript
// 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) {
    // Error handling
  }
```

Amazon Redshift examples
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](https://github.com).

### 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.

```javascript
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](https://github.com).

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 { ModifyClusterCommand } from '@aws-sdk/client-redshift';
import { redshiftClient } from './libs/redshiftClient.js';

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.

```javascript
// 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.

```javascript
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](https://github.com).

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.

```javascript
// 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 () => {

Amazon Redshift examples

124
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.

```javascript
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 `IDENTITY_TYPE` with the identity type, which can be "EmailAddress" or "Domain".

```javascript
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](https://github.com/awsdocs/aws-sdk-for-javascript-v3/tree/master/guides/examples).

**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.

```javascript
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](https://github.com).

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.

```javascript
// 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.

```bash
$ node ses_verifyemailidentity.js
```

### Note

Replace **ADDRESS@DOMAIN.EXT** with the email address, such as name@example.com.
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.

```javascript
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](https://github.com/aws-samples/aws-sdk-for-javascript/tree/main/email-samples/verify-email-identity).

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](https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/).
import { VerifyDomainIdentityCommand } from '@aws-sdk/client-ses';
import {
    getUniqueName,
    postfix,
} from '@aws-sdk-examples/libs/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.

```javascript
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](https://github.com/awsdocs/aws-sdk-js-docs/tree/master/examples/SES).

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.

Replace `IDENTITY_TYPE` with the identity type to be deleted, and `IDENTITY_NAME` with the name of the identity to be deleted.

```javascript
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.

```javascript
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](https://github.com).

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.

```javascript
import { ListTemplatesCommand } from '@aws-sdk/client-ses';
```

### 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 `ITEMS_COUNT` with the maximum number of templates to return. The value must be a minimum of 1 and a maximum of 10.
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 };
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.

```javascript
import { GetTemplateCommand } from '@aws-sdk/client-ses';
import { getUniqueName } from '@aws-sdk-examples/libs/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 (err) {
        console.log("Failed to get email template.", err);
        return err;
    }
};
```

To run the example, enter the following at the command prompt. Amazon SES returns the template details.
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.

```javascript
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 };
```

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
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, `HTML_CONTENT` with the HTML tagged content of email, `SUBJECT` with the subject of the email, and `TEXT_CONTENT` with the text of the email.

```javascript
import { CreateTemplateCommand } from '@aws-sdk/client-ses';
import { sesClient } from './libs/sesClient.js';
import { getUniqueName } from '@aws-sdk-examples/libs/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.

```bash
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.

```javascript
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, `HTML_CONTENT` with the HTML tagged content of email, `SUBJECT` with the subject of the email, and `TEXT_CONTENT` with the text of the email.

```javascript
import { UpdateTemplateCommand } from '@aws-sdk/client-ses';
import { getUniqueName } from '@aws-sdk-examples/libs/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.

```javascript
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 required TemplateName 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.

```javascript
code
import { DeleteTemplateCommand } from '@aws-sdk/client-ses';
import { getUniqueName } from '@aws-sdk-examples/libs/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.

```bash
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 TypeScrip 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.

```javascript
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](https://github.com/awsdocs/aws-sdk-for-javascript-dev-guide).

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.

```javascript
import { SendEmailCommand } from '@aws-sdk/client-ses';
import { sesClient } from './libs/sesClient.js';

const createSendEmailCommand = (toAddress, fromAddress) => {
  return new SendEmailCommand({
    // Amazon SES examples
    ReceiverAddresses: [toAddress],
    Source: fromAddress,
    // Email parameters
    Message: {
      Subject: {
        Data: "Your Subject",
       Charset: "UTF-8"
      },
      Body: {
        Text: {
          Data: "Your body in plain text",
          Charset: "UTF-8"
        },
        Html: {
          Data: "Your body in HTML",
          Charset: "UTF-8"
        }
      }
    }
  });
}
```

### 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](https).

### Note

Replace `RECEIVER_ADDRESS` with the address to send the email to, and `SENDER_ADDRESS` with the email address to the send the email from.

```javascript
import { SendEmailCommand } from '@aws-sdk/client-ses';
import { sesClient } from './libs/sesClient.js';

const createSendEmailCommand = (toAddress, fromAddress) => {
  return new SendEmailCommand({
    // Email parameters
    Message: {
      Subject: {
        Data: "Your Subject",
       Charset: "UTF-8"
      },
      Body: {
        Text: {
          Data: "Your body in plain text",
         Charset: "UTF-8"
        },
        Html: {
          Data: "Your body in HTML",
         Charset: "UTF-8"
        }
      }
    }
  });
}
```
const run = async () => {
  const sendEmailCommand = createSendEmailCommand(
    "recipient@example.com",
    "sender@example.com",
  );

  try {
    return await sesClient.send(sendEmailCommand);
  } catch (e) {
    // Handle error
  }
};
```javascript
console.error("Failed to send email.");
return e;
}
};
```

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](https://github.com).

## 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](https://aws.amazon.com).  

### Note

Replace `REGION` with your AWS Region, `RECEIVER_ADDRESS` with the address to send the email to, `SENDER_ADDRESS` with the email address to the 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-sdk-examples/libs/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({
    Destination: { ToAddresses: [user.emailAddress] },
    TemplateData: JSON.stringify({ contact: { firstName: user.firstName } }),
    Source: VERIFIED_EMAIL,
    Template: templateName,
  });
};

const run = async () => {
  const sendReminderEmailCommand = createReminderEmailCommand(
    USER,
    Amazon SES examples
    149
try {
  return await sesClient.send(sendReminderEmailCommand);
} catch (err) {
  console.log("Failed to send template email", err);
  return err;
}

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.

```javascript
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.

Amazon SES examples
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 RECEIVER_ADDRESSES with the address to send the email to, and SENDER_ADDRESS with the email address to send the email from.

```javascript
import { SendBulkTemplatedEmailCommand } from '@aws-sdk/client-ses';
import {
  getUniqueName,
  postfix,
} from '@aws-sdk-examples/libs/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
```
const createBulkReminderEmailCommand = (users, templateName) => {
  return new SendBulkTemplatedEmailCommand({
    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 (err) {
    console.log("Failed to send bulk template email", err);
    return err;
  }
};

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`. 
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.

```javascript
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](https://github.com).

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.

```javascript
import { CreateTopicCommand } from '@aws-sdk/client-sns';
import { snsClient } from '../libs/snsClient.js';

/**
 * @param {string} topicName - The name of the topic to create.
 */
```

Note

Replace `TOPIC_NAME` with the name of the topic.
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
  //   },
  // }
  return response;
};

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

```shell
node create-topic.js
```

This example code can be found [here on GitHub](https://github.com).

### 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.

```javascript
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](https://github.com).
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).

```javascript
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](https://github.com).

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.

```javascript
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
    // }}

Note: Replace `TOPIC_ARN` with the Amazon Resource Name (ARN) of the topic you are deleting.
```
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.

```
import { GetTopicAttributesCommand } from '@aws-sdk/client-sns';

Note
Replace `TOPIC_ARN` with the ARN of the topic.
```
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: 'xxxxxxxxxxxx',
  //     SubscriptionsPending: '1',
  //     TracingConfig: 'PassThrough',
  //     EffectiveDeliveryPolicy: '{"http":{"defaultHealthyRetryPolicy":
  //       "http":{"defaultHealthyRetryPolicy":
  //         {"minDelayTarget":20,"maxDelayTarget":20,"numRetries":3,"numMaxDelayRetries":0,"numNoDelayRetries":0,"numMinDelayRetries":0,"backoffFunction":"linear"},"disableSubscriptionOverrides":false,"defaultRequestPolicy":
  //         {"headerContentType":"text/plain; charset=UTF-8"}}},
  //       {"http":{"defaultRequestPolicy":
  //         {"headerContentType":"text/plain; charset=UTF-8"}}},
  //       {"http":{"defaultRequestPolicy":
  //         {"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.

```javascript
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](https://github.com/). 

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.

```
import { SetTopicAttributesCommand } from '@aws-sdk/client-sns';
import { snsClient } from '../libs/snsClient.js';

export const setTopicAttributes = async (topicArn = "TOPIC_ARN",
  ATTRIBUTE_NAME = "ATTRIBUTE_NAME",
  NEW_ATTRIBUTE_VALUE = "NEW_ATTRIBUTE_VALUE")
  => {
    return new SetTopicAttributesCommand({
      TopicArn: topicArn,
      AttributeName: ATTRIBUTE_NAME,
      NewAttributeValue: NEW_ATTRIBUTE_VALUE
    }).promise();

setTopicAttributes();
```

**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.
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.

```javascript
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](https://github.com).

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.

```javascript
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,
        })
    );
}
```

Note: Replace `MESSAGE_TEXT` with the message text, and `TOPIC_ARN` with the ARN of the SNS topic.
To run the example, enter the following at the command prompt.

```
node publish-topic.js
```

This example code can be found [here on GitHub](https://github.com).

### 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](https://github.com).

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.

```javascript
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': '0934fedef-0c4b-572e-9ed2-a3e38fadb0c8',
    //     'extendedRequestId': undefined,
    ```

Note

Replace `TOPIC_ARN` with the Amazon Resource Name (ARN) for the topic whose subscriptions you want to list.
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](https://github.com/aws-samples/aws-sdk-js-demos/tree/main/sns-subscribe).

### 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.

```javascript
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](https://github.com/aws-samples/aws-sdk-js-demos/tree/main/sns-subscribe).

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](https://github.com/aws-samples/aws-sdk-js-demos/tree/main/sns-subscribe).

### 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.

```javascript
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](https://github.com/aws-samples/aws-sdk-js-demos/tree/main/sns-subscribe).
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.

```javascript
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': {
```
To run the example, enter the following at the command prompt.

```
node subscribe-email.js
```

This example code can be found [here on GitHub](https://github.com/aws-samples/amazon-sns).

**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.

```javascript
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](https://github.com/aws-samples/amazon-sns).

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`.  

Amazon SNS Examples
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.

```javascript
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(new ConfirmSubscriptionCommand({
    Token: token,
    TopicArn: topicArn,
  }));
  // A subscription only needs to be confirmed if the endpoint type is HTTP/S, email, or in another AWS account.
  // If this is true, the subscriber cannot unsubscribe while unauthenticated.
```
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.

```javascript
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.

```
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,
    //   }
    // }
```
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.

```javascript
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](https://github.com/aws/aws-sdk-js/blob/master/docs/examples/sns-client.js).

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.

```javascript
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'
  // }
};
```

### 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.
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.

```javascript
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.
```javascript
import { UnsubscribeCommand } from '@aws-sdk/client-sns';
import { snsClient } from '../libs/snsClient.js';

/**
 * @param {string} subscriptionArn - The ARN of the subscription to cancel.
 */
  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](https://github.com/aws/aws-sdk-js-client-v3/tree/master/examples/unsubscribe).

**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.

```javascript
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.

```javascript
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.

```bash
node get-sms-attributes.js
```

This example code can be found [here on GitHub](https://github.com/aws/aws-sdk-js-sdk-locales/blob/master/docs/zh-hans/build/sns-messaging-attributes.js).
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.

```javascript
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.

```javascript
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.

```javascript
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.

```javascript
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
    // }
```
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](https://github.com).

### 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.

```javascript
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](https://github.com).

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.

```javascript
import { ListPhoneNumbersOptedOutCommand } from '@aws-sdk/client-sns';
import { snsClient } from '../libs/snsClient.js';

export const listPhoneNumbersOptedOut = async () => {
  const response = await snsClient.send(
    new ListPhoneNumbersOptedOutCommand({}),
  );
  return response;
};
```
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.

```javascript
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.

```
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": {
```
To run the example, enter the following at the command prompt.

```
node publish-sms.js
```

This example code can be found [here on GitHub](https://github.com).

### 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.

```javascript
const { TranscribeClient } = require('@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](https://github.com/awslabs/aws-sdk-js-SDK-3).

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.

```javascript
// 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"
};
```

**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.
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.

```
const { TranscribeClient } = require("@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.

```javascript
// 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](https://github.com).
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.

```javascript
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](https://github.com/awslabs/samples).

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.

```javascript
// 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");
  } catch (error) {
    console.error("Error - unable to delete:", error)
  }
};
```

**Note**

Replace `JOB_NAME` with the name of the job to delete.

Amazon Transcribe examples
To run the example, enter the following at the command prompt.

```bash
code
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 TypeScrip 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).
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.

```javascript
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.

- 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.

```javascript
// 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.

```javascript
const { TranscribeClient } = require("@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](https://github.com/awslabs/aws-sdk-js-examples/tree/master/mediarecognition).

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.

```
// 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.
  }
};
```

Note: Replace `KEYWORD` with a keyword that the returned jobs name must contain.
To run the example, enter the following at the command prompt.

```bash
node transcribe-list-medical-jobs.js
```

This sample code can be found [here on GitHub](https://github.com).

### 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.

```javascript
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](https://github.com).

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 for Linux Instances*.

⚠️ **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 for Linux Instances*.

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.

```bash
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.5/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. Activate `nvm` by typing the following at the command line.
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](https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/creating-an-amazon-ami.html) in the *Amazon EC2 User Guide for Linux Instances*.  

---
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](https://github.com).

- Create a shared configurations file with your user credentials. For more information about providing a shared credentials file, see [Shared config and credentials files](https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/shared-credentials-file.html) in the **AWS SDKs and Tools Reference Guide**. 

---

Cross-service: App to submit data

202
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.
For more information on the `create-stack` command parameters, see the [AWS CLI Command Reference guide](https://aws-docs.github.io/AWSCLICommandReference/command-reference.html) and the [AWS CloudFormation User Guide](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/).

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.

```javascript
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](https://github.com/awsdocs/aws-sdk-js-v3/tree/main/docs/examples/cloudformation).
Next, create a `dynamoAppHelperFiles` folder in your project folder, create a file `update-table.js` in it, and copy the content [here on GitHub](https://github.com) into it.

```javascript
// 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](https://github.com).

**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](https://github.com). 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](https://github.com/awslabs/aws-sdk-js-v3/tree/master/docs/examples/).  

### 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.

```javascript
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](https://github.com/awslabs/aws-sdk-js-v3/tree/master/docs/examples/).

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`.

```javascript
// 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.');
    }
  }
};
```
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, ak standard local formatted number, such as (415) 555-2671, is +14155552671 in E.164
    // format, where '1' in 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 no added to table
    console.error("An error occurred. Check the console for further information",
        err,
    );
}
// Display alert if all field 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:

```bash
webpack add_data.js --mode development --target web --devtool false -o main.js
```

Note
For information about installing webpack, see Bundling 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](#).

**Build a transcription app with authenticated users**

In this tutorial, you learn how to:

- Implement authentication using an Amazon Cognito identity pool to accept users federated with a Amazon Cognito user pool.
- Use Amazon Transcribe to transcribe and display voice recordings in the browser.

**The scenario**

The app enables users to sign up with a unique email and username. On confirmation of their email, they can record voice messages that are automatically transcribed and displayed in the app.

**How it works**

The app uses two Amazon S3 buckets, one to host the application code, and another to store transcriptions. The app uses an Amazon Cognito user pool to authenticate your users. Authenticated users have IAM permissions to access the required AWS services.

The first time a user records a voice message, Amazon S3 creates a unique folder with the user's name in the Amazon S3 bucket for storing transcriptions. Amazon Transcribe transcribes the voice message to text, and saves it in JSON in the user's folder. When the user refreshes the app, their transcriptions are displayed and available for downloading or deletion.
The tutorial should take about 30 minutes to complete.

**Steps**

**To build the app:**

1. [Prerequisites](#)
2. [Create the AWS resources](#)
3. [Create the HTML](#)
4. [Prepare the browser script](#)
5. [Run the app](#)
6. [Delete the resources](#)

**Prerequisites**

- Set up the project environment to run this Node JavaScript 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 section describes how to provision AWS resources for this app using the AWS Cloud Development Kit (AWS CDK).
The AWS CDK is a software development framework that enables you to define cloud application resources. For more information, see the AWS Cloud Development Kit (AWS CDK) Developer Guide.

To create resources for the app, use the template here on GitHub to create a AWS CDK stack using either the AWS Web Services Management Console or the AWS CLI. For instructions on how to modify the stack, or to delete the stack and its associated resources when you have finished the tutorial, see here on GitHub.

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.

The resulting stack automatically provisions the following resources.

- An Amazon Cognito identity pool with an authenticated user role.
- An IAM policy with permissions for the Amazon S3 and Amazon Transcribe is attached to the authenticated user role.
- An Amazon Cognito user pool that enables users to sign up and sign in to the app.
- An Amazon S3 bucket to host the application files.
- An Amazon S3 bucket to store the transcriptions.

This Amazon S3 bucket allows READ (LIST) public access, which enables anyone to list the objects within the bucket and potentially misuse the information. If you do not delete this Amazon S3 bucket immediately after completing the tutorial, we highly recommend you comply with the Security Best Practices in Amazon S3 in the Amazon Simple Storage Service User Guide.
Create the HTML

Create an index.html file, and copy and paste the content below into it. The page features panel of buttons for recording voice messages, and a table displaying the current user’s previously transcribed messages. The script tag at the end of the body element invokes the main.js, which contain all the browser script for the app. You create the main.js using Webpack, as described in the following section of this tutorial.

```html
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>title</title>
    <link rel="stylesheet" type="text/css" href="recorder.css">
    <style>
        table, td {
            border: 1px solid black;
        }
    </style>
</head>
<body>
<h2>Record</h2>
<p>
    <button id="record" onclick="startRecord()"></button>
    <button id="stopRecord" disabled onclick="stopRecord()">Stop</button>
</p>
<p>
    <audio id="recordedAudio"></audio>
</p>
<h2>My transcriptions</h2>
<table id="myTable1" style="width:678px;">
</table>
<table id="myTable" style="width:678px;">
<tr>
    <td style = "font-weight:bold">Time created</td>
    <td style = "font-weight:bold">Transcription</td>
    <td style = "font-weight:bold">Download</td>
    <td style = "font-weight:bold">Delete</td>
</tr>
</table>
```
This code example is available [here on GitHub](https://github.com/aws-samples/aws-sdk-for-javascript).

## Prepare the browser script

There are three files, `index.html`, `recorder.js`, and `helper.js`, which you are required to bundle into a single `main.js` using Webpack. This section describes in detail only the functions in `index.js` that use the SDK for JavaScript, which is available [here on GitHub](https://github.com/aws-samples/aws-sdk-for-javascript).

### Note

`recorder.js` and `helper.js` are required but, because they do not contain Node.js code, are explained in the inline comments [here](https://github.com/aws-samples/aws-sdk-for-javascript) and [here](https://github.com/aws-samples/aws-sdk-for-javascript) respectively GitHub.

First, define the parameters. `COGNITO_ID` is the endpoint for the Amazon Cognito User Pool you created in the [Create the AWS resources](https://github.com/aws-samples/aws-sdk-for-javascript) topic of this tutorial. It is formatted `cognito-idp.AWS_REGION.amazonaws.com/USER_POOL_ID`. The user pool id is `ID_TOKEN` in the AWS credentials token, which is stripped from the app URL by the `getToken` function in the 'helper.js' file. This token is passed to the `loginData` variable, which provides the Amazon Transcribe and Amazon S3 client objects with logins. Replace "REGION" with the AWS Region, and "BUCKET" with the 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.

```javascript
// Import the required AWS SDK clients and commands for Node.js
import "./helper.js";
import "./recorder.js";
import { CognitoIdentityClient } from "@aws-sdk/client-cognito-identity";
import { fromCognitoIdentityPool } from "@aws-sdk/credential-provider-cognito-identity";
import { CognitoIdentityProviderClient, GetUserCommand } from "@aws-sdk/client-cognito-identity-provider";
import { S3RequestPresigner } from "@aws-sdk/s3-request-presigner";
```

Cross-service: Transcription app
import { createRequest } from "@aws-sdk/util-create-request";
import { formatUrl } from "@aws-sdk/util-format-url";
import {
    TranscribeClient,
    StartTranscriptionJobCommand,
} from "@aws-sdk/client-transcribe";
import {
    S3Client,
    PutObjectCommand,
    GetObjectCommand,
    ListObjectsCommand,
    DeleteObjectCommand,
} from "@aws-sdk/client-s3";
import fetch from "node-fetch";

// Set the parameters.
// 'COGINTO_ID' has the format 'cognito-idp.eu-west-1.amazonaws.com/COGNITO_ID'.
let COGINTO_ID = "COGNITO_ID";
// Get the Amazon Cognito ID token for the user. 'getToken()' is in 'helper.js'.
let idToken = getToken();
let loginData = {
    [COGINTO_ID]: idToken,
};

const params = {
    Bucket: "BUCKET", // The Amazon Simple Storage Solution (S3) bucket to store the transcriptions.
    Region: "REGION", // The AWS Region
    identityPoolID: "IDENTITY_POOL_ID", // Amazon Cognito Identity Pool ID.
};

// Create an Amazon Transcribe service client object.
const client = new TranscribeClient({
    region: params.Region,
    credentials: fromCognitoIdentityPool({
        client: new CognitoIdentityClient({
            region: params.Region,
        }),
        identityPoolId: params.identityPoolID,
        logins: loginData,
    }),
});

// Create an Amazon S3 client object.
const s3Client = new S3Client({
    region: params.Region,
});
When the HTML page loads, the updateUserInterface creates a folder with the user's name in the Amazon S3 bucket if it's the first time they've signed in to the app. If not, it updates the user interface with any transcripts from the user's previous sessions.

```javascript
window.onload = async () => {
    // Set the parameters.
    const userParams = {
        // Get the access token. 'GetAccessToken()' is in 'helper.js'.
        AccessToken: getAccessToken(),
    };
    // Create a CognitoIdentityProviderClient client object.
    const client = new CognitoIdentityProviderClient({ region: params.Region });
    try {
        const data = await client.send(new GetUserCommand(userParams));
        const username = data.Username;
        // Export username for use in 'recorder.js'.
        exports.username = username;
        try {
            // If this is user's first sign-in, create a folder with user's name in Amazon S3 bucket.
            // Otherwise, no effect.
            const Key = `${username}/`;
            try {
                const data = await s3Client.send(
                    new PutObjectCommand({ Key: Key, Bucket: params.Bucket })
                );
                console.log("Folder created for user ", data.Username);
            } catch (err) {
                console.log("Error", err);
            }
            try {
                // Get a list of the objects in the Amazon S3 bucket.
                const data = await s3Client.send(
                );
            } catch (err) {
                console.log("Error", err);
            }
        } catch (err) {
            console.log("Error", err);
        }
    } catch (err) {
        console.log("Error", err);
    }
};
```
// Create a variable for the list of objects in the Amazon S3 bucket.
const output = data.Contents;
// Loop through the objects, populating a row on the user interface for each object.
for (var i = 0; i < output.length; i++) {
    var obj = output[i];
    const objectParams = {
        Bucket: params.Bucket,
        Key: obj.Key,
    };
    // Get the name of the object from the Amazon S3 bucket.
    const data = await s3Client.send(new GetObjectCommand(objectParams));
    // Extract the body contents, a readable stream, from the returned data.
    const result = data.Body;
    // Create a variable for the string version of the readable stream.
    let stringResult = "";
    // Use 'yieldUint8Chunks' to convert the readable streams into JSON.
    for await (let chunk of yieldUint8Chunks(result)) {
        stringResult += String.fromCharCode.apply(null, chunk);
    }
    // The setTimeout function waits while readable stream is converted into JSON.
    setTimeout(function () {
        // Parse JSON into human readable transcript, which will be displayed on user interface (UI).
        const outputJSON = JSON.parse(stringResult).results.transcripts[0].transcript;
        // Create name for transcript, which will be displayed.
        const outputJSONTime = JSON.parse(stringResult).jobName.split("/") [0]
            .replace("-job", "");
        i++;
        // Display the details for the transcription on the UI.
        // 'displayTranscriptionDetails()' is in 'helper.js'.
        displayTranscriptionDetails(
            i,
            outputJSONTime,
            objectParams.Key,
            outputJSON
        );
        }, 1000);
    }
} catch (err) {
```javascript
// Convert readable streams.
async function* yieldUint8Chunks(data) {
    const reader = data.getReader();
    try {
        while (true) {
            const { done, value } = await reader.read();
            if (done) return;
            yield value;
        }
    } finally {
        reader.releaseLock();
    }
}
```

When the user records a voice message for transcriptions, the upload uploads the recordings to the Amazon S3 bucket. This function is called from the recorder.js file.

```javascript
// Upload recordings to Amazon S3 bucket
window.upload = async function (blob, userName) {
    // Set the parameters for the recording recording.
    const Key = `${userName}/test-object-${Math.ceil(Math.random() * 10 ** 10)}`;
    let signedUrl;

    // Create a presigned URL to upload the transcription to the Amazon S3 bucket when it is ready.
    try {
        // Create an Amazon S3RequestPresigner object.
        const signer = new S3RequestPresigner({ ...s3Client.config });
        // Create the request.
        const request = await createRequest(
            s3Client,
```
new PutObjectCommand({ Key, Bucket: params.Bucket });

// Define the duration until expiration of the presigned URL.
const expiration = new Date(Date.now() + 60 * 60 * 1000);

// Create and format the presigned URL.
signedUrl = formatUrl(await signer.presign(request, expiration));

console.log(`\nPutting "${Key}"`);

} catch (err) {
  console.log("Error creating presigned URL", err);
}

try {
  // Upload the object to the Amazon S3 bucket using a presigned URL.
  response = await fetch(signedUrl, {
    method: "PUT",
    headers: {
      "content-type": "application/octet-stream",
    },
    body: blob,
  });

  // Create the transcription job name. In this case, it's the current date and time.
  const today = new Date();
  const date =
    today.getFullYear() + 
    "-" +
    (today.getMonth() + 1) +
    "-" +
    today.getDate();
  const time =
    today.getHours() + "-" + today.getMinutes() + "-" + today.getSeconds();
  const jobName = date + "-time-" + time;

  // Call the "createTranscriptionJob()" function.
  createTranscriptionJob(
    "s3://" + params.Bucket + "/" + Key, 
    jobName, 
    params.Bucket,
    Key 
  );

} catch (err) {
  console.log("Error uploading object", err);
}

// Create the AWS Transcribe transcription job.
const createTranscriptionJob = async (recording, jobName, bucket, key) => {
  // Set the parameters for transcriptions job
  const params = {
    TranscriptionJobName: jobName + "-job",
    LanguageCode: "en-US", // For example, 'en-US',
    OutputBucketName: bucket,
    OutputKey: key,
    Media: {
      MediaFileUri: recording, // For example, "https://transcribe-demo.s3-REGION.amazonaws.com/hello_world.wav"
    },
  }
  try {
    // Start the transcription job.
    const data = await client.send(new StartTranscriptionJobCommand(params));
    console.log("Success - transcription submitted", data);
  } catch (err) {
    console.log("Error", err);
  }
};

deleteTranscription deletes a transcription from the user interface, and deleteRow deletes an existing transcription from the Amazon S3 bucket. Both are triggered by the Delete button on the user interface.

// Delete a transcription from the Amazon S3 bucket.
window.deleteJSON = async (jsonFileName) => {
  try {
    await s3Client.send(
      new DeleteObjectCommand({
        Bucket: params.Bucket,
        Key: jsonFileName,
      })
    );
    console.log("Success - JSON deleted");
  } catch (err) {
    console.log("Error", err);
  }
};

// Delete a row from the user interface.
window.deleteRow = function (rowid) {
  const row = document.getElementById(rowid);
Finally, run the following at the command prompt to bundle the JavaScript for this example in a file named main.js:

```
webpack index.js --mode development --target web --devtool false -o main.js
```

**Note**
For information about installing webpack, see [Bundling applications with webpack](#).

### Run the app

You can view the app at the location below.

```
DOMAIN/login?
client_id=APP_CLIENT_ID&response_type=token&scope=aws.cognito.signin.user.admin+email+openid+phone+profile&redirect_uri=REDIRECT_URL
```

Amazon Cognito makes it easy to run the app by providing a link in the AWS Web Services Management Console. Simply navigate to the App client setting of your Amazon Cognito user pool, and select the **Launch Hosted UI**. The URL for the app has the following format.

**Important**
The Hosted UI defaults to a response type of 'code'. However, this tutorial is designed for the 'token' response type, so you have to change it.

### Delete the AWS resources

When you finish the tutorial, you should delete the resources so you do not incur any unnecessary charges. Because you added content to both Amazon S3 buckets, you must delete them manually. Then you can delete the remaining resources using either the [AWS Web Services Management Console](#) or the [AWS CLI](#). Instructions on how to modify the stack, or to delete the stack and its associated resources when you have finished the tutorial, see [here on GitHub](#).
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.

---

**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

This topic is part of a tutorial that demonstrates how to invoke a Lambda function through Amazon API Gateway using the AWS SDK for JavaScript. To start at the beginning of the tutorial, see Invoking Lambda with API Gateway.

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 topic is part of a tutorial that demonstrates how to invoke a Lambda function through Amazon API Gateway using the AWS SDK for JavaScript. To start at the beginning of the tutorial, see Invoking Lambda with API Gateway.

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.

   ```bash
   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.

```javascript
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](https://github.com).

Next, create a file named `populate-table.js` in the root directory of your project folder, and copy the content [here on GitHub](https://github.com) 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.

```bash
node populate-table.js
```

```javascript
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: "155555555555654" },
                    startDate: { S: "2019-12-20" },
                },
            },
            
            PutRequest: {
                Item: {
                    id: { N: "2" },
                    firstName: { S: "Xing" },
                    phone: { N: "155555555555653" },
                    startDate: { S: "2019-12-17" },
                },
            },
        ],
    },
};
```
```javascript
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](https://github.com).

### 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.

```javascript
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](https://github.com).
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.)

```javascript
// 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 Bundling applications with webpack.

2. Run the following in the command line to bundle the JavaScript for this example into a file called <index.js>:

   ```bash
   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

This topic is part of a tutorial that demonstrates how to invoke a Lambda function through Amazon API Gateway using the AWS SDK for JavaScript. To start at the beginning of the tutorial, see Invoking Lambda with API Gateway.

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.

```javascript
// 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 the 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.

```bash
node lambda-function-setup.ts
```

This code example is available [here on GitHub](#).
Configure API Gateway to invoke the Lambda function

This topic is part of a tutorial that demonstrates how to invoke a Lambda function through Amazon API Gateway using the AWS SDK for JavaScript. To start at the beginning of the tutorial, see Invoking Lambda with API Gateway.

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 screen](image)

4. Specify Employee as the API name and provide a description.
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**.
10. Choose /employees, select Create Method from the Actions, then select GET from the drop-down 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**.
3. Fill in the **Deploy API** form and choose **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.
Delete the resources

This topic is part of a tutorial that demonstrates how to invoke a Lambda function through Amazon API Gateway using the AWS SDK for JavaScript. To start at the beginning of the tutorial, see Invoking Lambda with API Gateway.

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.

For more AWS cross-service examples, see AWS SDK for JavaScript cross-service examples.

Creating AWS serverless workflows using AWS SDK for JavaScript

You can create an AWS serverless workflow by using Step Functions the AWS SDK for Java and AWS Step Functions. Each workflow step is implemented using an AWS Lambda function. Lambda is a compute service that enables you to run code without provisioning or managing servers. Step Functions is a serverless orchestration service that lets you combine Lambda functions and other AWS services to build business-critical applications.
Note
You can create Lambda functions in various programming languages. For this tutorial, Lambda functions implemented by using the Lambda Java API. For more information about Lambda, see What is Lambda.

In this tutorial, you create a workflow that creates support tickets for an organization. Each workflow step performs an operation on the ticket. This tutorial shows you how to use JavaScript to process workflow data. For example, you'll learn how to read data that's passed to the workflow, how to pass data between steps, and how to invoke AWS services from the workflow.

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 no longer charged.

To build the app:

1. Prerequisite tasks
2. Create the AWS resources
3. Creating the workflow
4. Create the Lambda functions
5. Execute your workflow by using the Step Functions console

Prerequisite tasks

This topic is part of a tutorial that demonstrates to to invoke Lambda functions using AWS Step Functions. To start at the beginning of the tutorial, see Creating AWS serverless workflows using AWS SDK for JavaScript.

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 topic is part of a tutorial that demonstrates how to invoke Lambda functions using AWS Step Functions. To start at the beginning of the tutorial, see Creating AWS serverless workflows using AWS SDK for JavaScript.

This tutorial requires the following resources.

- An Amazon DynamoDB table named Case with a key named Id.
- An IAM role named lambda-support used to invoke Lambda functions. This role has policies that enable it to invoke the Amazon DynamoDB and Amazon Simple Email Service services from a Lambda function.
- An IAM role named workflow-support used to invoke the workflow.
- An Amazon S3 bucket to host the Lambda functions.

You can create these resources manually, but we recommend provisioning these resources using the AWS Cloud Development Kit (AWS CDK) (AWS CDK) as described in this tutorial.

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.
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.

```bash
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.

Create the AWS resources using the Amazon Web Services Management Console;

To create resources for the app in the console, follow the instructions in the AWS CloudFormation User Guide. Use the template provided create a file named `setup.yaml`, and copy the content here on GitHub.

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.

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.

Creating the workflow

This topic is part of a tutorial that demonstrates to to invoke Lambda functions using AWS Step Functions. To start at the beginning of the tutorial, see Creating AWS serverless workflows using AWS SDK for JavaScript.

The following figure shows the workflow you’ll create with this tutorial.
The following is what happens at each step in the workflow:

+ **Start** - Initiates the workflow.

+ **Open Case** – Handles a support ticket ID value by passing it to the workflow.

+ **Assign Case** – Assigns the support case to an employee and stores the data in a DynamoDB table.

+ **Send Email** – Sends the employee an email message by using the Amazon Simple Email Service (Amazon SES) to inform them there is a new ticket.

+ **End** - Stops the workflow.

**Create a serverless workflow by using Step functions**

You can create a workflow that processes support tickets. To define a workflow by using Step Functions, you create an Amazon States Language (JSON-based) document to define your state.
machine. An Amazon States Language document describes each step. After you define the
document, Step functions provides a visual representation of the workflow. The following figure
shows the Amazon States Language document and the visual representation of the workflow.

Workflows can pass data between steps. For example, the Open Case step processes a case ID
value (passed to the workflow) and passes that value to the Assign Case step. Later in this tutorial,
you’ll create application logic in the Lambda function to read and process the data values.

To create a workflow

1. Open the Amazon Web Services Console.
2. Choose Create State Machine.
3. Choose Author with code snippets. In the Type area, choose Standard.

4. Specify the Amazon States Language document by entering the following code.

```json
{
   "Comment": "A simple AWS Step Functions state machine that automates a call center
   support session.",
   "StartAt": "Open Case",
   "States": {
      "Open Case": {
         "Type": "Task",
         "Next": "Assign Case"
      },
      "Assign Case": {
         "Type": "Task",
```
"Next": "Send Email"
},
"Send Email": {
"Type": "Task",
"End": true
}
}

Note

Don’t worry about the errors related to the Lambda resource values. You’ll update these values later in this tutorial.

5. Choose Next.

6. In the name field, enter SupportStateMachine.

7. In the Permission section, choose Choose an existing role.

8. Choose workflow-support (the IAM role that you created).

9. Choose Create state machine. A message appears that states the state machine was successfully created.
Create the Lambda functions

This topic is part of a tutorial that demonstrates how to invoke Lambda functions using AWS Step Functions. To start at the beginning of the tutorial, see Creating AWS serverless workflows using AWS SDK for JavaScript.

Use the Lambda runtime API to create the Lambda functions. In this example, there are three workflow steps that each correspond to each Lambda function.

Create these Lambda function, as described in the following sections:

- **getId Lambda function** - Used as the first step in the workflow that processes the ticket ID value.
- **addItem Lambda class** - Used as the second step in the workflow that assigns the ticket to an employee and stores the data in a DynamoDB database.
- **sendemail Lambda class** - Used as the third step in the workflow that uses the Amazon SES to send an email message to the employee to notify them about the ticket.

**getId Lambda function**

Create a Lambda function that returns the ticket ID value that is passed to the second step in the workflow.

```javascript
exports.handler = async (event) => {
```
// Create a support case using the input as the case ID, then return a confirmation message
try{
    const myCaseID = event.inputCaseID;
    var myMessage = "Case " + myCaseID + ": opened...";
    var result = { Case: myCaseID, Message: myMessage };
} catch(err){
    console.log('Error', err);
}

Enter the following in the command line to use webpack to bundle the file into a file named
index.js.

webpack getid.js --mode development --target node --devtool false --output-library-target umd -o index.js

Then compress index.js into a ZIP file name getid.js.zip. Upload the ZIP file to the
Amazon S3 bucket you created in the the topic of this example.

This code example is available here on GitHub.

addItem Lambda class

Create a Lambda function that selects an employee to assign the ticket, then stores the ticket data
in a DynamoDB table named Case.

"use strict";
// Load the required clients and commands.
const { PutItemCommand } = require ( "@aws-sdk/client-dynamodb" );
const { dynamoClient } = require ( "../libs/dynamoClient" );

exports.handler = async (event) => {
    try {
        // Helper function to send message using Amazon SNS.
        const val = event;
        //PersistCase adds an item to a DynamoDB table
        const tmp = Math.random() <= 0.5 ? 1 : 2;
        console.log(tmp);
        if (tmp == 1) {
            const params = {
                (null)
            };
Enter the following in the command line to use webpack to bundle the file into a file named index.js.
webpack additem.js --mode development --target node --devtool false --output-library-target umd -o index.js

Then compress index.js into a ZIP file name additem.js.zip. Upload the ZIP file to the Amazon S3 bucket you created in the topic of this example.

This code example is available [here on GitHub](https://github.com).

**sendemail Lambda class**

Create a Lambda function that sends an email to notify them about the new ticket. The email address that is passed from the second step is used.

```javascript
// Load the required clients and commands.
const { SendEmailCommand } = require('@aws-sdk/client-ses');
const { sesClient } = require('../libs/sesClient');

exports.handler = async (event) => {
    // Enter a sender email address. This address must be verified.
    const senderEmail = "SENDER_EMAIL"
    const sender = "Sender Name <" + senderEmail + ">";

    // AWS Step Functions passes the employee's email to the event.
    // This address must be verified.
    const recepient = event.S;

    // The subject line for the email.
    const subject = "New case";

    // The email body for recipients with non-HTML email clients.
    const body_text = "Hello,
    Please check the database for new ticket assigned to you.";

    // The HTML body of the email.
    const body_html = `<html><head></head><body>Hello!<h1>Please check the database for new ticket assigned to you.</h1></body></html>`;

    // The character encoding for the email.
    const charset = "UTF-8";

    var params = {
        Source: sender,
        Destination: {
            Cross-service: Serverless workflows with Step Functions
        }
    };

```
ToAddresses: [recepient],
},
Message: {
    Subject: {
        Data: subject,
        CharSet: charset,
    },
    Body: {
        Text: {
            Data: body_text,
            CharSet: charset,
        },
        Html: {
            Data: body_html,
            CharSet: charset,
        },
    },
},
};

try {
    const data = await sesClient.send(new SendEmailCommand(params));
    console.log(data);
} catch (err) {
    console.error(err);
}

Enter the following in the command line to use webpack to bundle the file into a file named index.js.

```bash
webpack sendemail.js --mode development --target node --devtool false --output-library-target umd -o index.js
```

Then compress index.js into a ZIP file name sendemail.js.zip. Upload the ZIP file to the Amazon S3 bucket you created in the the topic of this example.

This code example is available here on GitHub.

**Deploy the Lambda functions**

To deploy the getid Lambda function:

1. Open the Lambda console at Amazon Web Services Console.
2. Choose Create Function.
3. Choose Author from scratch.
4. In the Basic information section, enter `getid` as the name.
5. In the Runtime, choose Node.js 14x.
6. Choose Use an existing role, and then choose lambda-support (the IAM role that you created in the).
7. Choose Create function.
8. choose Upload from - Amazon S3 location.
9. Choose Upload, choose Upload from - Amazon S3 location, and enter the Amazon S3 link URL.
10. Choose Save.
11. Repeat this procedure for the `additem.js.zip` and `sendemail.js.zip` to new Lambda functions. When you finish, you will have three Lambda functions that you can reference in the Amazon States Language document.

Add the Lambda functions to workflows

This topic is part of a tutorial that demonstrates to to invoke Lambda functions using AWS Step Functions. To start at the beginning of the tutorial, see Creating AWS serverless workflows using AWS SDK for JavaScript.

1. Open the Lambda console. Notice that you can view the Lambda Amazon Resource Name (ARN) value in the upper-right corner.
2. Copy the value and then paste it into step 1 of the Amazon States Language document, located in the Step Functions console.

3. Update the Resource for the Assign Case and Send Email steps. This is how you hook in Lambda functions created by using the AWS SDK for Java into a workflow created by using Step Functions.

**Execute your workflow by using the Step Functions console**

This topic is part of a tutorial that demonstrates how to invoke Lambda functions using AWS Step Functions. To start at the beginning of the tutorial, see [Creating AWS serverless workflows using AWS SDK for JavaScript](#).

You can invoke the workflow on the Step Functions console. An execution receives JSON input. For this example, you can pass the following JSON data to the workflow.

```json
{
  "inputCaseID": "001"
}
```

To execute your workflow:

1. On the Step Functions console, choose **Start execution**.

2. In the **Input** section, pass the JSON data. View the workflow. As each step is completed, it turns green.
3. If the step turns red, an error occurred. You can click the step and view the logs that are accessible from the right side.
When the workflow is finished, you can view the data in the DynamoDB table.
Congratulations, you have created an AWS serverless workflow by using the AWS SDK for Java. 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.

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 a 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.

![DynamoDB Table Screenshot]

**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

This topic is part of a tutorial that demonstrates how to invoke a Lambda function using Amazon CloudWatch scheduled events using the AWS SDK for JavaScript. To start at the beginning of the tutorial, see Creating scheduled events to execute AWS Lambda functions.

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 topic is part of a tutorial that demonstrates how to invoke a Lambda function through Amazon CloudWatch scheduled events using the AWS SDK for JavaScript. To start at the beginning of the tutorial, see Creating scheduled events to execute AWS Lambda functions.

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](https://github.com) into it.

   **Note**
   
   The AWS CloudFormation template was generated using the AWS CDK available [here on GitHub](https://github.com). 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.

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

   **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.

   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.

Cross-service: Scheduled Lambda events
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.

```javascript
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.
dynamoClient = new DynamoDBClient({region:REGION});
module.exports = { dynamoClient };
```

This code is available [here on GitHub](https).

Next, create a file named `populate-table.js` in the root directory of your project folder, and copy the content [here on GitHub](https) 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
```

```javascript
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" },
                    },
                },
            }]
        }
    }
};
```

Cross-service: Scheduled Lambda events
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.)

```javascript
"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.)

```javascript
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 mylambdafnction.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 Bundling applications with webpack.
2. Run the following in the command line to bundle the JavaScript for this example into a file called `<index.js>`:

```bash
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`.

```javascript
"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);
  }
};
AWS SDK for JavaScript

Developer Guide for SDK Version 3

Deploy the Lambda function
This topic is part of a tutorial that demonstrates how to invoke a Lambda function through
Amazon CloudWatch scheduled events using the AWS SDK for JavaScript. To start at the beginning
of the tutorial, see Creating scheduled events to execute AWS Lambda functions.
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 name 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/v3lambda-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.",
};
Cross-service: Scheduled Lambda events

262


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 *).

Note
For more information, see Using Lambda with CloudWatch Events.
Delete the resources

This topic is part of a tutorial that demonstrates how to invoke a Lambda function through Amazon CloudWatch scheduled events using the AWS SDK for JavaScript. To start at the beginning of the tutorial, see Creating scheduled events to execute AWS Lambda functions.

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.

For more AWS cross-service examples, see AWS SDK for JavaScript cross-service examples.

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.

I need a hotel room

What city will you be staying in?

Ottawa Canada

What day do you want to check in?

Monday

How many nights will you be staying?

2

What type of room would you like, queen, king or deluxe?

King

Okay, I have you down for a 2 night stay in Ottawa Canada starting 2021-01-25. Shall I book the reservation?

J’ai besoin d’une chambre d’hôtel

Send Text

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.
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 (version 3) 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](https://aws.amazon.com/free/).

**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

This topic is part of a tutorial that create an Amazon Lex chatbot within a web application to engage your web site visitor. To start at the beginning of the tutorial, see Building an Amazon Lex chatbot.

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 topic is part of a tutorial that create an Amazon Lex chatbot within a web application to engage your web site visitor. To start at the beginning of the tutorial, see Building an Amazon Lex chatbot.

This tutorial requires the following resources.

- An unauthenticated IAM role with attached permissions to:
  - Amazon Comprehend
  - Amazon Translate
• Amazon Lex

You can create this 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 from GitHub into it.

   **Note**

   The AWS CloudFormation template was generated using the AWS CDK available from 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.

   ```bash
   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 Amazon Lex console to create the bot. This example 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).
• 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)  

• 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**

This topic is part of a tutorial that create an Amazon Lex chatbot within a web application to engage your web site visitor. To start at the beginning of the tutorial, see **Building an Amazon Lex chatbot**.
Create a file named `index.html`. Copy and paste the code below into `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.

```html
<!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/">
      Amazon Lex (product)
    </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" />
  <button onclick="createResponse()">Send Text</button>
  <script type="text/javascript" src="./main.js"></script>
</body>
```

This code is also available [here on GitHub](https://github.com/aws-samples/amazon-lex-booktrip).
Create the browser script

This topic is part of a tutorial that create an Amazon Lex chatbot within a web application to engage your web site visitor. To start at the beginning of the tutorial, see Building an Amazon Lex chatbot.

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.

```javascript
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.](https://github.com/aws/aws-sdk-js-v3)
AWS SDK for JavaScript

Developer Guide for SDK Version 3

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);
Cross-service: Amazon Lex example

275


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 Bundling 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:

```bash
webpack index.js --mode development --target web --devtool false -o main.js
```

Next steps

This topic is part of a tutorial that create an Amazon Lex chatbot within a web application to engage your web site visitor. To start at the beginning of the tutorial, see Building an Amazon Lex chatbot.

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.
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](https://aws.amazon.com/free/).

**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**

This topic is part of a tutorial that create an AWS application that sends and retrieves messages by using the AWS SDK for JavaScript and Amazon Simple Queue Service (Amazon SQS). To start at the beginning of the tutorial, see [Creating an example messaging application](#).

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](https://github.com).
- Create a shared configurations file with your user credentials. For more information about providing a shared credentials file, see [Shared config and credentials files](https://aws.amazon.com) in the [AWS SDKs and Tools Reference Guide](https://aws.amazon.com).
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 topic is part of a tutorial that create an AWS application that sends and retrieves messages by using the AWS SDK for JavaScript and Amazon Simple Queue Service (Amazon SQS). To start at the beginning of the tutorial, see Creating an example messaging application.

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 this 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](https://github.com) into it.

   **Note**
   The AWS CloudFormation template was generated using the AWS CDK available [here on GitHub](https://github.com). 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.

   ```bash
   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](https://aws.amazon.com), 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**

This topic is part of a tutorial that create an AWS application that sends and retrieves messages by using the AWS SDK for JavaScript and Amazon Simple Queue Service (Amazon SQS). To start at the beginning of the tutorial, see [Creating an example messaging application](https://aws.amazon.com).

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**

This topic is part of a tutorial that creates an AWS application that sends and retrieves messages by using the AWS SDK for JavaScript and Amazon Simple Queue Service (Amazon SQS). To start at the beginning of the tutorial, see [Creating an example messaging application](#).

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 into `index.html`. This HTML references `main.js`. This is a bundled version of `index.js`, which includes the required AWS SDK for JavaScript modules.

```html
<!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>
</head>
```
<div class="container">
  <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>
Cross-service: Messaging app

<!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox --><!-- All of these child items are hidden and only displayed in a FancyBox--
This code is also available [here on GitHub](https://github.com).

**Creating the browser script**

This topic is part of a tutorial that creates an AWS application that sends and retrieves messages by using the AWS SDK for JavaScript and Amazon Simple Queue Service (Amazon SQS). To start at the beginning of the tutorial, see [Creating an example messaging application](#).

In this topic, you create 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](https://github.com) 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 file named `sqsClient.js`. Replace `REGION` and `IDENTITY_POOL_ID` in each.

```javascript
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,
});
```

Note

Use the ID of the Amazon Cognito identity pool you created in [Create the AWS resources](#).
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.

```javascript
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.

```javascript
$(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 = '
            '<img src='' +
            image_url +
            '' alt="Avatar" class="right" style="width:100%;''">';
        myTextNode.html(userText);
        myTextNode.append(images_div);

        // Add the message to the GUI.
        $('#messages').append(myTextNode);
    }
} catch (err) {
    console.log("Error loading messages: ", 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.

```javascript
const pushMessage = async () => {
    // Get and convert user and message input.
    var user = $('input[name=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",
                },
            },
        };
    } catch (err) {
        console.log("Error retrieving SQS queue URL: ", err);
    }
};
```
```javascript
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 = "<img src="" +
    image_url +
    " alt="Avatar" class="right" style="width:100%;"/>";
    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 Bundling 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>:

   ```shell
   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.

For more AWS cross-service examples, see AWS SDK for JavaScript cross-service examples.
Using 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.

**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.

   ```bash
   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.)

   ```bash
   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.
   ```

**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](https://github.com/nvm-sh/nvm) (Node Version Manager) GitHub repository.
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:

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

Note
For more information about Webpack, see Bundling 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.

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 and cross-service examples.

Scenarios are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.

Cross-service examples are sample applications that work across multiple AWS services.

Examples

- Actions and scenarios using SDK for JavaScript (v3)
- Cross-service examples using SDK for JavaScript (v3)

Actions and scenarios using SDK for JavaScript (v3)

The following code examples show how to perform actions and implement common scenarios by using the AWS SDK for JavaScript (v3) with AWS services.

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 and cross-service examples.

Scenarios are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.

Services

- Auto Scaling 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)
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 and cross-service examples.

Scenarios are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.

Each example includes a link to GitHub, where you can find instructions on how to set up and run the code in context.

Topics
- Actions
- Scenarios

Actions

Attach an ELB target group to an Auto Scaling group

The following code example shows how to attach an ELB target group to an Auto Scaling group.

SDK for JavaScript (v3)

```javascript
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,
```
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,
import { ElasticLoadBalancingV2Client, waitUntilLoadBalancerAvailable, } from "@aws-sdk/client-elastic-load-balancing-v2";
import { ScenarioOutput, ScenarioInput, ScenarioAction, } from "@aws-sdk-examples/libs/scenario/index.js";
import { retry } from "@aws-sdk-examples/libs/utils/util-timers.js";
import { MESSAGES, NAMES, RESOURCES_PATH, ROOT } from "./constants.js";
import { initParamsSteps } from "./steps-reset-params.js";

/**
 * @type {import('@aws-sdk-examples/libs/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",
      },
    ),
  }),
];
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 },
})),
},
}}),

Auto Scaling
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("createdInstanceRole", () => {
  const client = new IAMClient({});
  return client.send(
    new CreateRoleCommand({
      RoleName: NAMES.instanceRoleName,
      AssumeRolePolicyDocument: readFileSync(
        join(ROOT, "assume-role-policy.json"),
      ),
    })),
  );
});
}

new ScenarioAction("attachPolicyToRole", async (state) => {
  const client = new IAMClient({});
  await client.send(
    new AttachRolePolicyCommand({
      RoleName: NAMES.instanceRoleName,
      PolicyArn: state.instancePolicyArn,
    })),
  );
});

Auto Scaling
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_PROFILE_ARN}", state.instanceProfileArn),
),
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 () => {
    // snippet-start:[javascript.v3.wkflw.resilient.CreateLaunchTemplate]
    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_start.sh"),
                    ).toString("base64"),
                    KeyName: NAMES.keyPairName,
                }
            }));
    // snippet-end:[javascript.v3.wkflw.resilient.CreateLaunchTemplate]
    new ScenarioOutput(
        "createdLaunchTemplate",
        MESSAGES.createdLaunchTemplate.replace("${LAUNCH_TEMPLATE_NAME}"),
    ),
});
new ScenarioAction("createAutoScalingGroup", async (state) => {
  const ec2Client = new EC2Client({});
  const { AvailabilityZones } = await ec2Client.send(
    new DescribeAvailabilityZonesCommand({}),
  );
  state.availabilityZoneNames = AvailabilityZones.map((az) => az.ZoneName);
  const autoScalingClient = new AutoScalingClient({});
  await retry({ intervalInMs: 1000, maxRetries: 30 }, () =>
    autoScalingClient.send(
      new CreateAutoScalingGroupCommand({
        AvailabilityZones: state.availabilityZoneNames,
        AutoScalingGroupName: NAMES.autoScalingGroupName,
        LaunchTemplate: {
          LaunchTemplateName: NAMES.launchTemplateName,
          Version: "$Default",
        },
        MinSize: 3,
        MaxSize: 3,
      }),
    ),
  );
  state.availabilityZoneNames = state.availabilityZoneNames.map((az) => az.ZoneName);

  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) => {
    // snippet-start:[javascript.v3.wkflw.resilient.DescribeVpcs]
    const client = new EC2Client({});
    const { Vpcs } = await client.send(
        new DescribeVpcsCommand(
            Filters: [{ Name: "is-default", Values: ["true"] }],
        ),
    );
    // snippet-end:[javascript.v3.wkflw.resilient.DescribeVpcs]
    state.defaultVpc = Vpcs[0].VpcId;
}),
new ScenarioOutput("gotVpc", (state) =>
    MESSAGES.gotVpc.replace("${VPC_ID}", state.defaultVpc),
),
new ScenarioOutput("gettingSubnets", MESSAGES.gettingSubnets),
new ScenarioAction("getSubnets", async (state) => {
    // snippet-start:[javascript.v3.wkflw.resilient.DescribeSubnets]
    const client = new EC2Client({});
    const { Subnets } = await client.send(
        new DescribeSubnetsCommand(
            Filters: [
                { Name: "vpc-id", Values: [state.defaultVpc] },
                { Name: "availability-zone", Values: state.availabilityZoneNames },
                { Name: "default-for-az", Values: ["true"] },
            ],
        ),
    );
    // snippet-end:[javascript.v3.wkflw.resilient.DescribeSubnets]
    state.subnets = Subnets.map((subnet) => subnet.SubnetId);
}),
new ScenarioOutput("gotSubnets",
/**
 * @param {{ subnets: string[] }} state
 */
(state) =>
    MESSAGES.gotSubnets.replace("${SUBNETS}", state.subnets.join(" ")),
),
new ScenarioOutput(
   "creatingLoadBalancerTargetGroup",
   MESSAGES.creatingLoadBalancerTargetGroup.replace(
      "${TARGET_GROUP_NAME}"
      NAMES.loadBalancerTargetGroupName,
   ),
),
new ScenarioAction("createLoadBalancerTargetGroup", async (state) => {
   // snippet-start:[javascript.v3.wkflw.resilient.CreateTargetGroup
   const client = new ElasticLoadBalancingV2Client({});
   const { TargetGroups } = await client.send(
      new CreateTargetGroupCommand({
         Name: NAMES.loadBalancerTargetGroupName,
         Protocol: "HTTP",
         Port: 80,
         HealthCheckPath: "/healthcheck",
         HealthCheckIntervalSeconds: 10,
         HealthCheckTimeoutSeconds: 5,
         HealthyThresholdCount: 2,
         UnhealthyThresholdCount: 2,
         VpcId: state.defaultVpc,
      }),
   );
   // snippet-end:[javascript.v3.wkflw.resilient.CreateTargetGroup
   const targetGroup = TargetGroups[0];
   state.targetGroupArn = targetGroup.TargetGroupArn;
   state.targetGroupProtocol = targetGroup.Protocol;
   state.targetGroupPort = targetGroup.Port;
}),
new ScenarioOutput(
   "createdLoadBalancerTargetGroup",
   MESSAGES.createdLoadBalancerTargetGroup.replace(
      "${TARGET_GROUP_NAME}"
      NAMES.loadBalancerTargetGroupName,
   ),
),
new ScenarioOutput(
   "creatingLoadBalancer",
   MESSAGES.creatingLoadBalancer.replace("${LB_NAME}", NAMES.loadBalancerName),
),
new ScenarioAction("createLoadBalancer", async (state) => {
   // snippet-start:[javascript.v3.wkflw.resilient.CreateLoadBalancer
   const client = new ElasticLoadBalancingV2Client({});
   const { LoadBalancers } = await client.send(
      new CreateLoadBalancerCommand({
         LoadBalancerName: NAMES.loadBalancerName,
         Scheme: "internet-facing",
         VpcId: state.defaultVpc,
      }),
   );
   // snippet-end:[javascript.v3.wkflw.resilient.CreateLoadBalancer
   const loadBalancer = LoadBalancers[0];
   state.loadBalancerArn = loadBalancer.LoadBalancerArn;
   state.loadBalancerName = loadBalancer.LoadBalancerName;
   state.loadBalancerScheme = loadBalancer.Scheme;
   state.loadBalancerPort = loadBalancer.Port;
});
new CreateLoadBalancerCommand({
    Name: NAMES.loadBalancerName,
    Subnets: state.subnets,
  })),
);  
state.loadBalancerDns = LoadBalancers[0].DNSName;
state.loadBalancerArn = LoadBalancers[0].LoadBalancerArn;
await waitUntilLoadBalancerAvailable(
    { client },
    { Names: [NAMES.loadBalancerName] },
  );
// snippet-end:[javascript.v3.wkflw.resilient.CreateLoadBalancer]
}),
new ScenarioOutput("createdLoadBalancer", (state) =>
  MESSAGES.createdLoadBalancer
    .replace("${LB_NAME}", NAMES.loadBalancerName)
    .replace("${DNS_NAME}", state.loadBalancerDns),
),
new ScenarioOutput("creatingListener",
  MESSAGES.creatingLoadBalancerListener
    .replace("${LB_NAME}", NAMES.loadBalancerName)
    .replace("${TARGET_GROUP_NAME}", NAMES.loadBalancerTargetGroupName),
),
new ScenarioAction("createListener", async (state) => {
  // snippet-start:[javascript.v3.wkflw.resilient.CreateListener]
  const client = new ElasticLoadBalancingV2Client({});
  const { Listeners } = await client.send(
    new CreateListenerCommand({
      LoadBalancerArn: state.loadBalancerArn,
      Protocol: state.targetGroupProtocol,
      Port: state.targetGroupPort,
      DefaultActions: [
        { Type: "forward", TargetGroupArn: state.targetGroupArn },
      ],
    })),
  // snippet-end:[javascript.v3.wkflw.resilient.CreateListener]
  const listener = Listeners[0];
  state.loadBalancerListenerArn = listener.ListenerArn;
});
new ScenarioOutput("createdListener", (state) =>
  MESSAGES.createdLoadBalancerListener.replace("${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) => {
  // snippet-start:[javascript.v3.wkflw.resilient.AttachTargetGroup]
  const client = new AutoScalingClient({});
  await client.send(
    new AttachLoadBalancerTargetGroupsCommand({
      AutoScalingGroupName: NAMES.autoScalingGroupName,
      TargetGroupARNs: [state.targetGroupArn],
    }),
  );
  // snippet-end:[javascript.v3.wkflw.resilient.AttachTargetGroup]
}),
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];
```javascript
/**
 * @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 {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 {any[]} state
 */
(state) => {
  if (state.myIpRules.length > 0) {
    return false;
  } else {
    return MESSAGES.noIpRules;
  }
},
);```
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) {
      // Handle error
    }
  })
Create steps to run the demo.

/∗
∗ Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
∗ SPDX-License-Identifier: Apache-2.0
/∗
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-sdk-examples/libs/scenario/scenario.js";  
import { retry } from "@aws-sdk-examples/libs/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:
${JSON.stringify(state.recommendation, null, 2)}',
        { preformatted: true },
    });

const getHealthCheck = new ScenarioAction("getHealthCheck", async (state) => {
    // snippet-start:[javascript.v3.wkflw.resilient.DescribeTargetGroups]
    const client = new ElasticLoadBalancingV2Client({});
    const { TargetGroups } = await client.send(
        new DescribeTargetGroupsCommand({
            Names: [NAMES.loadBalancerTargetGroupName],
        })),
    );
    // snippet-end:[javascript.v3.wkflw.resilient.DescribeTargetGroups]

    // snippet-start:[javascript.v3.wkflw.resilient.DescribeTargetHealth]
    const { TargetHealthDescriptions } = await client.send(
        new DescribeTargetHealthCommand({
            TargetGroupArn: TargetGroups[0].TargetGroupArn,
        })),
    );
    // snippet-end:[javascript.v3.wkflw.resilient.DescribeTargetHealth]
    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}

`; +
            return `Health check:
${status}`;
        },
        { preformatted: true },
    );

const loadBalancerLoop = new ScenarioAction(}
"loadBalancerLoop",
getRecommendation.action,
{
  whileConfig: {
    inputEquals: true,
    input: new ScenarioInput(
      "loadBalancerCheck",
      MESSAGES.demoLoadBalancerCheck,
      {
        type: "confirm",
      },
      ),
    output: getRecommendationResult,
  },
},
);

const healthCheckLoop = new ScenarioAction(
  "healthCheckLoop",
  getHealthCheck.action,
  {
    whileConfig: {
      inputEquals: true,
      input: new ScenarioInput("healthCheck", MESSAGES.demoHealthCheck, {
        type: "confirm",
      }),
    output: getHealthCheckResult,
    },
  },
  );

const statusSteps = [
  getRecommendation,
  getRecommendationResult,
  getHealthCheck,
  getHealthCheckResult,
];

/**
 * @type {import('@aws-sdk-examples/libs/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,
            }),
        );
    }
});
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 },
    }),
  ),
);

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) {
    // Auto Scaling
  }
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,
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",
        }));
    // snippet-start:[javascript.v3.wkflw.resilient.CreateInstanceProfile]
    const { InstanceProfile } = await iamClient.send(
        new CreateInstanceProfileCommand({
            InstanceProfileName: NAMES.ssmOnlyInstanceProfileName,
        }));
    // snippet-end:[javascript.v3.wkflw.resilient.CreateInstanceProfile]
Create steps to destroy all of the resources.

```javascript
import { unlinkSync } from "node:fs";
import { DynamoDBClient, DeleteTableCommand } from "@aws-sdk/client-dynamodb";
import { EC2Client, DeleteKeyPairCommand, DeleteLaunchTemplateCommand, } from "@aws-sdk/client-ec2";
import { IAMClient, DeleteInstanceProfileCommand, RemoveRoleFromInstanceProfileCommand, DeletePolicyCommand, DeleteRoleCommand, DetachRolePolicyCommand, paginateListPolicies, } from "@aws-sdk/client-iam";
import { AutoScalingClient, } from "@aws-sdk/client-autoscaling";
```
import { ScenarioOutput, ScenarioInput, ScenarioAction, } from "@aws-sdk-examples/libs/scenario/index.js";
import { retry } from "@aws-sdk-examples/libs/utils/util-timers.js";
import { MESSAGES, NAMES } from "./constants.js";
import { findLoadBalancer } from "./shared.js";

/** *
 * @type {import('@aws-sdk-examples/libs/scenario.js').Step[]} export const destroySteps = [
 * 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}",
 * 
 * )
 * *
 * )
 * */

export const destroySteps = [
    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("${TABLE_NAME}",
        })
    });
new ScenarioAction("deleteKeyPair", async (state) => {
    try {
        const client = new EC2Client({});
        await client.send(new DeleteKeyPairCommand({ KeyName: NAMES.keyPairName }));
        unlinkSync(`${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 ${NAMES.instancePolicyName} not found.`);
        } else {
            await client.send(new DetachRolePolicyCommand({
                RoleName: NAMES.instanceRoleName,
            }));
        }
    } catch (e) {
        state.detachPolicyFromRoleError = e;
    }
}),
new ScenarioOutput("detachPolicyFromRoleResult", (state) => {
    if (state.detachPolicyFromRoleError) {
        console.error(state.detachPolicyFromRoleError);
        return MESSAGES.detachPolicyFromRoleError.replace("${ROLE_NAME}", NAMES.instanceRoleName, );
    } else {
        return MESSAGES.deletedPolicyFromRole.replace("${ROLE_NAME}", NAMES.instanceRoleName, );
    }
})}
PolicyArn: policy.Arn,
}),
);
} catch (e) {
  state.detachPolicyFromRoleError = e;
}
}),
new ScenarioOutput("detachedPolicyFromRole", (state) => {
  if (state.detachPolicyFromRoleError) {
    console.error(state.detachPolicyFromRoleError);
    return MESSAGES.detachedPolicyFromRoleError
      .replace("${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.deletePolicyResult.replace(
      "$\{INSTANCE\_POLICY\_NAME\}$
    , NAMES.instancePolicyName,
    );
  }};
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 {
        // snippet-start:[javascript.v3.wkflw.resilient.DeleteInstanceProfile]
        const client = new IAMClient({});
        await client.send(
            new DeleteInstanceProfileCommand({
                InstanceProfileName: NAMES.instanceProfileName,
            }),
        );
        // snippet-end:[javascript.v3.wkflw.resilient.DeleteInstanceProfile]
    } 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("deleteLaunchTemplate", async (state) => {
    const client = new EC2Client({});
    try {
        // snippet-start:[javascript.v3.wkflw.resilient.DeleteLaunchTemplate]
        const client = new EC2Client({});
        await client.send(
            new DeleteLaunchTemplateCommand({
                LaunchTemplateName: NAMES.launchTemplateName,
            }),
        );
        // snippet-end:[javascript.v3.wkflw.resilient.DeleteLaunchTemplate]
        const client = new EC2Client({});
        await client.send(
            new DeleteLaunchTemplateCommand({
                LaunchTemplateName: NAMES.launchTemplateName,
            }),
        );
        // snippet-end:[javascript.v3.wkflw.resilient.DeleteLaunchTemplate]
    } catch (e) {
        state.deleteLaunchTemplateError = e;
    }
});
// snippet-start:[javascript.v3.wkflw.resilient.DeleteLaunchTemplate]
await client.send(
    new DeleteLaunchTemplateCommand({
        LaunchTemplateName: NAMES.launchTemplateName,
    }),
); // snippet-end:[javascript.v3.wkflw.resilient.DeleteLaunchTemplate]

try {
    await terminateGroupInstances(NAMES.autoScalingGroupName);
    await retry({ intervalInMs: 30000, 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("deleteAutoScalingGroup", async (state) => {
try {
    await terminateGroupInstances(NAMES.autoScalingGroupName);
    await retry({ intervalInMs: 30000, 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("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) => {
  try {
    const client = new ElasticLoadBalancingV2Client({});
    
    // snippet-start:[javascript.v3.wkflw.resilient.DeleteTargetGroup]
    const targetGroup = await findTargetGroup(NAMES.targetGroupName);
    const targetGroups = 
      await client.send(new DescribeTargetGroupsCommand({
        TargetGroupArns: [targetGroup.TargetGroupArn,]
      }));
    await retry({ intervalInMs: 1000, maxRetries: 60 }, async () => {
      const targetGroup = 
        await findTargetGroup(NAMES.targetGroupName);
      if (targetGroup) {
        throw new Error("Target group still exists.");
      }
    });
  } catch (e) {
    state.deleteLoadBalancerTargetGroupError = e;
  }
  
},
new ScenarioOutput("deleteLoadBalancerTargetGroupResult", (state) => {
  if (state.deleteLoadBalancerTargetGroupError) {
    console.error(state.deleteLoadBalancerTargetGroupError);
    return MESSAGES.deleteLoadBalancerTargetGroupError.replace("${TG_NAME}"
      , NAMES.targetGroupName,
    );
  } else {
    return MESSAGES.deletedLoadBalancerTargetGroup.replace("${TG_NAME}"
      , NAMES.targetGroupName,
    );
  }
})}
,
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;
} // snippet-end:[javascript.v3.wkflw.resilient.DeleteTargetGroup]
),
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: ssmOnlyPolicy.Arn,
      }));
  } catch (e) {
    state.detachSsmOnlyAWSRolePolicyError = e;
  }
})};
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,
  );

Auto Scaling
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);
  } else {
    return MESSAGES.deletedSsmOnlyRole.replace(
      "${POLICY_NAME}",
      NAMES.ssmOnlyRoleName,
    );
  }
},
new ScenarioAction("deleteSsmOnlyAttachment", async (state) => {
  try {
    const iamClient = new IAMClient({});
    await iamClient.send(
      new DeleteRoleCommand(
        {RoleName: NAMES.ssmOnlyRoleName},
      ),
    );
  } catch (e) {
    state.deleteSsmOnlyAttachmentError = e;
  }
},
new ScenarioOutput("deleteSsmOnlyAttachmentResult", (state) => {
  if (state.deleteSsmOnlyAttachmentError) {
    console.error(state.deleteSsmOnlyAttachmentError);
  } else {
    return MESSAGES.deletedSsmOnlyAttachment.replace(
      "${POLICY_NAME}",
      NAMES.ssmOnlyRoleName,
    );
  }
},
new ScenarioAction("createSsmOnlyRole", async (state) => {
  try {
    const iamClient = new IAMClient({});
    await iamClient.send(
      new CreateRoleCommand(
        {RoleName: NAMES.ssmOnlyRoleName},
      ),
    );
  } catch (e) {
    state.createSsmOnlyRoleError = e;
  }
},
new ScenarioOutput("createSsmOnlyRoleResult", (state) => {
  if (state.createSsmOnlyRoleError) {
    console.error(state.createSsmOnlyRoleError);
  } else {
    return MESSAGES.createdSsmOnlyRole.replace(
      "${POLICY_NAME}",
      NAMES.ssmOnlyRoleName,
    );
  }
},
new ScenarioAction("createSsmOnlyPolicy", async (state) => {
  try {
    const iamClient = new IAMClient({});
    await iamClient.send(
      new CreatePolicyCommand(
        {PolicyName: NAMES.ssmOnlyPolicyName},
      ),
    );
  } catch (e) {
    state.createSsmOnlyPolicyError = e;
  }
},
new ScenarioOutput("createSsmOnlyPolicyResult", (state) => {
  if (state.createSsmOnlyPolicyError) {
    console.error(state.createSsmOnlyPolicyError);
  } else {
    return MESSAGES.createdSsmOnlyPolicy.replace(
      "${POLICY_NAME}",
      NAMES.ssmOnlyPolicyName,
    );
  }
});
```javascript
return MESSAGES.deleteSsmOnlyRoleError.replace(
  "${ROLE_NAME}",
  NAMES.ssmOnlyRoleName,
); else {
  return MESSAGES.deletedSsmOnlyRole.replace(
    "${ROLE_NAME}",
    NAMES.ssmOnlyRoleName,
  );
} },
]
]

/**
 * @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;
    }
  }
```
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
• ReplaceIamInstanceProfileAssociation
• TerminateInstanceInAutoScalingGroup
• UpdateAutoScalingGroup

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 and cross-service examples.

Scenarios are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.

Each example includes a link to GitHub, where you can find instructions on how to set up and run the code in context.

Topics

- Actions

Actions

Create a metric alarm

The following code example shows how to create or update an Amazon CloudWatch alarm and associate it with the specified metric, metric math expression, anomaly detection model, or Metrics Insights query.

SDK for JavaScript (v3)

```javascript
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
        CloudWatchAlarmTarget: {
            MetricName: 'CPUUtilization',
            Namespace: 'AWS/EC2',
            Dimensions: [
                {Name: 'InstanceId', Value: 'i-0123456789abcdef0'},
            ],
            Statistic: 'Maximum',
            Period: 60,
            Threshold: 70,
            ComparisonOperator: 'GreaterThanThreshold',
        },
    });
    await client.send(command);
}
```

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.

```javascript
import { CloudWatchClient } from '@aws-sdk/client-cloudwatch';
import { DEFAULT_REGION } from '@aws-sdk-examples/libs/utils/util-aws-sdk.js';

export const client = new CloudWatchClient({ region: DEFAULT_REGION });
```

- For more information, see [AWS SDK for JavaScript Developer Guide](https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/)
// 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);
    }
});
Delete alarms

The following code example shows how to delete Amazon CloudWatch alarms.

**SDK for JavaScript (v3)**

```javascript
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.

• For more information, see [AWS SDK for JavaScript Developer Guide](https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/).

Note

There’s more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/awsdocs/aws-doc-sdk-examples).
```javascript
import { CloudWatchClient } from '@aws-sdk/client-cloudwatch';
import { DEFAULT_REGION } from '@aws-sdk-examples/libs/utils/util-aws-sdk.js';

export const client = new CloudWatchClient({ region: DEFAULT_REGION });
```

- 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.

```javascript
// 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](#).
Describe alarms for a metric

The following code example shows how to describe Amazon CloudWatch alarms for a metric.

SDK for JavaScript (v3)

```javascript
import { DescribeAlarmsCommand } from '@aws-sdk/client-cloudwatch';
import { client } from '../libs/client.js';

const run = async () => {
  try {
    console.log('Running', run.name, 'module...');
    return await client.send(command);
  } catch (err) {
    console.error(err);
  }
};

export default run();
```

Create the client in a separate module and export it.

```javascript
import { CloudWatchClient } from '@aws-sdk/client-cloudwatch';
import { DEFAULT_REGION } from '@aws-sdk-examples/libs/utils/util-aws-sdk.js';

export const client = new CloudWatchClient({ region: DEFAULT_REGION });
```

For API details, see [DeleteAlarms](https://aws.amazon.com) 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](https://aws.amazon.com).
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.

Disable alarm actions

The following code example shows how to disable Amazon CloudWatch alarm actions.
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.

```javascript
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.

```javascript
import { CloudWatchClient } from '@aws-sdk/client-cloudwatch';
import { DEFAULT_REGION } from '@aws-sdk-examples/libs/utils/util-aws-sdk.js';

export const client = new CloudWatchClient({ region: DEFAULT_REGION });
```

- For more information, see AWS SDK for JavaScript Developer Guide.
- For API details, see DisableAlarmActions in AWS SDK for JavaScript API Reference.
Import the SDK and client modules and call the API.

```javascript
// 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](https://aws.amazon.com/documentation/javascript/)
- For API details, see [DisableAlarmActions](https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/CloudWatch.html#disableAlarmActions-property) in [AWS SDK for JavaScript API Reference](https://aws.amazon.com/documentation/javascript/api_reference/).

### Enable alarm actions

The following code example shows how to enable Amazon CloudWatch alarm actions.
Import the SDK and client modules and call the API.

```javascript
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.

```javascript
import { CloudWatchClient } from '@aws-sdk/client-cloudwatch';
import { DEFAULT_REGION } from '@aws-sdk-examples/libs/utils/util-aws-sdk.js';

export const client = new CloudWatchClient({ region: DEFAULT_REGION });
```

- 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.

```javascript
// 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%,'
};

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](https://docs.aws.amazon.com/sdk-for-javascript/v3/developerguide/).
• For API details, see [EnableAlarmActions](https://docs.aws.amazon.com/sdk-for-javascript/v3/developerguide/cloudwatch-api.html#cloudwatchapi-enablealarmactions) in [AWS SDK for JavaScript API Reference](https://docs.aws.amazon.com/sdk-for-javascript/v3/developerguide/).

### List metrics

The following code example shows how to list the metadata for Amazon CloudWatch metrics. To get data for a metric, use the GetMetricData or GetMetricStatistics actions.

#### SDK for JavaScript (v3)

**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws/samples).

```javascript
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';
import { DEFAULT_REGION } from '@aws-sdk-examples/libs/utils/util-aws-sdk.js';

export const client = new CloudWatchClient({ region: DEFAULT_REGION });

• 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'});
```javascript
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](https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/)

### Put data into a metric

The following code example shows how to publish metric data points to Amazon CloudWatch.

**SDK for JavaScript (v3)**

**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws-samples/amazon-cloudwatch-getting-started).

Import the SDK and client modules and call the API.

```javascript
import { PutMetricDataCommand } from '@aws-sdk/client-cloudwatch';
import { client } from '../libs/client.js';

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

```javascript
import { CloudWatchClient } from "@aws-sdk/client-cloudwatch";
import { DEFAULT_REGION } from "@aws-sdk-examples/libs/utils/util-aws-sdk.js";

export const client = new CloudWatchClient({ region: DEFAULT_REGION });
```

- For more information, see [AWS SDK for JavaScript Developer Guide](https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/index.html).
- For API details, see [PutMetricData](https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/CloudWatch.html#PutMetricData-property) in [AWS SDK for JavaScript API Reference](https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/CloudWatch.html#PutMetricData-property).
// 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 and cross-service examples.

Scenarios are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.

Each example includes a link to GitHub, where you can find instructions on how to set up and run the code in context.

Topics

- Actions

Actions

Adding a target

The following code example shows how to add a target to an Amazon CloudWatch Events event.

SDK for JavaScript (v3)

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";
import { DEFAULT_REGION } from "@aws-sdk-examples/libs/utils/util-aws-sdk.js";

export const client = new CloudWatchEventsClient({ region: DEFAULT_REGION });

- For more information, see [AWS SDK for JavaScript Developer Guide](https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/).
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.

```javascript
// 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.

Create a scheduled rule

The following code example shows how to create an Amazon CloudWatch Events scheduled rule.
Import the SDK and client modules and call the API.

```javascript
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.

```javascript
import { CloudWatchEventsClient } from '@aws-sdk/client-cloudwatch-events';
import { DEFAULT_REGION } from '@aws-sdk-examples/libs/utils/util-aws-sdk.js';
```
export const client = new CloudWatchEventsClient({ region: DEFAULT_REGION });

- 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)

ℹ️ Note
There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```javascript
// 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.
Send events

The following code example shows how to send Amazon CloudWatch Events events.

SDK for JavaScript (v3)

**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.

```javascript
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.

```javascript
import { CloudWatchEventsClient } from '@aws-sdk/client-cloudwatch-events';
import { DEFAULT_REGION } from '@aws-sdk-examples/libs/utils/util-aws-sdk.js';

export const client = new CloudWatchEventsClient({ region: DEFAULT_REGION });
```

- For more information, see [AWS SDK for JavaScript Developer Guide](https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/).
- For API details, see [PutEvents](https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/cloudwatch-events.html#cloudwatch-events-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](https://github.com/aws-ia/ia-private-code-examples).

```javascript
// 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.

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 and cross-service examples.

*Scenarios* are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.

Each example includes a link to GitHub, where you can find instructions on how to set up and run the code in context.

**Topics**

- Actions

**Actions**

**Create a log group**

The following code example shows how to create a new CloudWatch Logs log group.
**SDK for JavaScript (v3)**

**Note**
There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws-samples/aws-code-examples).

```javascript
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](https://docs.aws.amazon.com/sdk-for-javascript/v3/api/latest/index.html#aws-sdk/CloudWatchLogsClient/createLogGroup) in [AWS SDK for JavaScript API Reference](https).

**Create a subscription filter**

The following code example shows how to create an Amazon CloudWatch Logs subscription filter.

**SDK for JavaScript (v3)**

**Note**
There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws-samples/aws-code-examples).

**CloudWatch Logs**
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 cwl = new AWS.CloudWatchLogs({apiVersion: '2014-03-28'});

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

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

• For more information, see AWS SDK for JavaScript Developer Guide.
• For API details, see PutSubscriptionFilter in AWS SDK for JavaScript API Reference.

Delete a log group

The following code example shows how to delete an existing CloudWatch Logs log group.

SDK for JavaScript (v3)

----- 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';
Delete a subscription filter

The following code example shows how to delete an Amazon CloudWatch Logs subscription filter.

SDK for JavaScript (v3)

```javascript
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 [DeleteLogGroup](aws-sdk) in [AWS SDK for JavaScript API Reference](aws-sdk).

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](aws-code-examples).
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](#).

### Describe existing subscription filters

The following code example shows how to describe Amazon CloudWatch Logs existing subscription filters.

#### SDK for JavaScript (v3)

```javascript
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)**

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

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

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

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

• For more information, see [AWS SDK for JavaScript Developer Guide](https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/).

• For API details, see `DescribeSubscriptionFilters` in *AWS SDK for JavaScript API Reference*.

**Describe log groups**

The following code example shows how to describe CloudWatch Logs log groups.
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.

Start a Live Tail session

The following code example shows how to start a Live Tail session for an existing log group/log stream.

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,
  filterPattern: filterPattern
);
try{
  const response = await client.send(command);
  handleResponseAsync(response);
} catch (err){
  // Pre-stream exceptions are captured here
Stop the Live Tail session after a period of time has elapsed.

```javascript
/* 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*.

## 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 and cross-service examples.

*Scenarios* are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.

Each example includes a link to GitHub, where you can find instructions on how to set up and run the code in context.

### Topics
- Actions

### Actions

#### Create a project

The following code example shows how to create an CodeBuild project.
Create a project.

```javascript
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: {
                type: ArtifactsType.S3,
                location: buildOutputBucket,
            },
            environment: {
                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': {
//       httpStatusCode: 200,
//       requestId: 'b428b244-777b-49a6-a48d-5dffedced8e7',
//       extendedRequestId: undefined,
//       cfId: undefined,
//       attempts: 1,
//       totalRetryDelay: 0
//     },
//     project: {
//       artifacts: {
//         encryptionDisabled: false,
//         location: 'xxxxxxxx-xxxxxxxx-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: {


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 and cross-service examples.

*Scenarios* are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.
Each example includes a link to GitHub, 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)**

```javascript
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**

**Confirm a user**

The following code example shows how to confirm an Amazon Cognito user.

**SDK for JavaScript (v3)**

```javascript
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](https://aws.github.io/aws-sdk-js-v3/docs/api-reference/index.html#CognitoIdentityProviderClient) in *AWS SDK for JavaScript API Reference*.

**Confirm an MFA device for tracking**

The following code example shows how to confirm an MFA device for tracking by Amazon Cognito.
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);
};


Get a token to associate an MFA application with a user

The following code example shows how to get a token to associate an MFA application with an Amazon Cognito user.

SDK for JavaScript (v3)

- Note
  There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws/aws-sdk-js-v3/tree/master/code-examples).
const associateSoftwareToken = (session) => {
  const client = new CognitoIdentityProviderClient({});
  const command = new AssociateSoftwareTokenCommand({
    Session: session,
  });

  return client.send(command);
};

- For API details, see [AssociateSoftwareToken](https://aws-sdk.github.io/aws-sdk-js-v3/APIReference/API_AssociateSoftwareToken.html) in *AWS SDK for JavaScript API Reference*.

### Get information about a user

The following code example shows how to get information about an Amazon Cognito user.

**SDK for JavaScript (v3)**

```javascript
const adminGetUser = ({ userPoolId, username }) => {
  const client = new CognitoIdentityProviderClient({});

  const command = new AdminGetUserCommand({
    UserPoolId: userPoolId,
    Username: username,
  });

  return client.send(command);
};
```


---

*Note*

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws/aws-sdk-js-v3).
List users

The following code example shows how to list Amazon Cognito users.

SDK for JavaScript (v3)

```javascript
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.

Resend a confirmation code

The following code example shows how to resend an Amazon Cognito confirmation code.

SDK for JavaScript (v3)

```javascript
const resendConfirmationCode = ({ clientId, username }) => {
```

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 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.

Respond to SRP authentication challenges

The following code example shows how to respond to Amazon Cognito SRP authentication challenges.

SDK for JavaScript (v3)

```
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,
    },
  });
```
Respond to an authentication challenge

The following code example shows how to respond to an Amazon Cognito authentication challenge.

SDK for JavaScript (v3)

```javascript
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 `RespondToAuthChallenge` 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*. 
Sign up a user

The following code example shows how to sign up a user with Amazon Cognito.

SDK for JavaScript (v3)

```javascript
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*.

Start authentication with a tracked device

The following code example shows how to start authentication with a device tracked by Amazon Cognito.
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*.

### Start authentication with administrator credentials

The following code example shows how to start authentication with Amazon Cognito and administrator credentials.

---

**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws-samples/aws-sdk-for-javascript-example).

---

**SDK for JavaScript (v3)**
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.

Verify an MFA application with a user

The following code example shows how to verify an MFA application with an Amazon Cognito user.

SDK for JavaScript (v3)

```javascript
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,
   UserId: userId,
   UserAttributes: attributes,
   UserPoolId: poolId,
   UserSub: fakeSub,
   UserType: 'user'
});

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)

For the best experience, clone the GitHub repository and run this example. The following code represents a sample of the full example application.

```javascript
import { log } from '@aws-sdk-examples/libs/utils/util-log.js';
import { signUp } from '../../../actions/sign-up.js';
import { FILE_USER_POOLS } from './constants.js';
import { getSecondValuesFromEntries } from '@aws-sdk-examples/libs/utils/util-csv.js';

const validateClient = (clientId) => {
  if (!clientId) {
    throw new Error('Amazon Cognito Identity Provider
');
```


`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 }],
  },
import { log } from "@aws-sdk-examples/libs/utils/util-log.js";
import { confirmSignUp } from "../../../actions/confirm-sign-up.js";
import { FILE_USER_POOLS } from "./constansts.js";
import { getSecondValuesFromEntries } from "@aws-sdk-examples/libs/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);
    /**
     *
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.');
}
}
}

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 "qrcode-terminal";
import { log } from "@aws-sdk-examples/libs/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-sdk-examples/libs/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'?\n');
  }
};

const validateId = (id) => {
  if (!id) {
    throw new Error('User pool id is missing. Did you run 'create-user-pool'?\n');
  }
};

const validateUser = (username, password) => {
  if (!(username && password)) {
    throw new Error('Username and password must be provided as arguments to the 'admin-initiate-auth' command.'\n);}
};

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-sdk-examples/libs/utils/util-log.js";
import { adminRespondToAuthChallenge } from "../../actions/admin-respond-to-auth-challenge.js";
import { getFirstEntry } from "@aws-sdk-examples/libs/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-responsd-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) => {
  try {
    const [_, username, totp] = commands;

    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-sdk-examples/libs/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
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.

*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 and cross-service examples.

*Scenarios* are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.

Each example includes a link to GitHub, 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](https://github.com).
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('
'));
    return response;
};

• For API details, see ListTables in AWS SDK for JavaScript API Reference.

Topics
• Actions
• Scenarios

Actions

Create a table

The following code example shows how to create a DynamoDB 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 { CreateTableCommand, DynamoDBClient } from '@aws-sdk/client-dynamodb';

const client = new DynamoDBClient({});
```js
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](https://aws.amazon.com/documentation/amazonjavascript/).
- For API details, see [CreateTable](https://aws.amazon.com/documentation/amazonjavascript/reference/)

**SDK for JavaScript (v2)**

ℹ️ **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws/aws-sdk-code-examples).
// 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);
    }
});
For more information, see AWS SDK for JavaScript Developer Guide.

For API details, see CreateTable in AWS SDK for JavaScript API Reference.

Delete a table

The following code example shows how to delete a DynamoDB table.

SDK for JavaScript (v3)

```javascript
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.
Delete an item from a table

The following code example shows how to delete an item from a DynamoDB table.
This example uses the document client to simplify working with items in DynamoDB. For API details see [DeleteCommand](#).

```javascript
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.

```javascript
// 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.

```javascript
// 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',
    Key: {
        'HASH_KEY': VALUE
    },
};
```
```javascript
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](#).

### Get a batch of items

The following code example shows how to get a batch of DynamoDB items.

#### SDK for JavaScript (v3)

---

**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](#).

```javascript
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.
      }
    }
  });
```

DynamoDB
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.

Get an item from a table

The following code example shows how to get an item from a DynamoDB 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.

This example uses the document client to simplify working with items in DynamoDB. For API details see GetCommand.

```javascript
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](https://aws-sdk-docs.github.io/aws-sdk-js-v3/api-docs/services/dynamodb.html#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](https://github.com/aws-samples/aws-code-examples).

Get an item from a table.

```javascript
// 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'}
  },
};
```
Get an item from a table using the DynamoDB document client.

```javascript
// 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](https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/).
Get information about a table

The following code example shows how to get information about a DynamoDB table.

SDK for JavaScript (v3)

```javascript
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)

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

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.
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.

List tables
The following code example shows how to list DynamoDB tables.

SDK for JavaScript (v3)

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";

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

    const response = await client.send(command);
    console.log(response);
    return response;
};
Put an item in a table

The following code example shows how to put an item in a DynamoDB table.

**SDK for JavaScript (v3)**

```javascript
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*. **Note**

  There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws/aws-sdk-js-3).
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.

```javascript
// 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.

```javascript
// 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 a table

The following code example shows how to query a DynamoDB 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.

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.

Run a PartiQL statement

The following code example shows how to run a PartiQL statement on a DynamoDB 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.

Create an item using PartiQL.

```javascript
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.

```javascript
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.

```javascript
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](https://docs.aws.amazon.com/sdk-for-javascript/v3/api/reference/DynamoDB.Client.html#DynamoDB.Client.ExecuteStatement) in *AWS SDK for JavaScript API Reference*.

**Run batches of PartiQL statements**

The following code example shows how to run batches of PartiQL statements on a DynamoDB table.

**SDK for JavaScript (v3)**

> ![](https://aws.amazon.com/badge/green.png) **Note**
> 
> There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws-code-examples).

Create a batch of items using PartiQL.

```javascript
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);
```
Get a batch of items using PartiQL.

```javascript
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.

```javascript
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=?",
      },
    ],
  });

  const response = await docClient.send(command);
  console.log(response);
  return response;
};
For API details, see [BatchExecuteStatement](https://docs.aws.amazon.com/AWSDynamoDB/latest/APIReference/API_BatchExecuteStatement.html) in *AWS SDK for JavaScript API Reference*.

### Scan a table

The following code example shows how to scan a DynamoDB table.

**SDK for JavaScript (v3)**

```javascript
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' },
        ...,
    });

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

This example uses the document client to simplify working with items in DynamoDB. For API details see [ScanCommand](https://docs.aws.amazon.com/AWSDynamoDB/latest/APIReference/API_Scan.html).

Note
---

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws-samples/aws-code-examples/tree/main/dynamodb/scan-a-table).

---

AWS SDK for JavaScript

Developer Guide for SDK Version 3

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](https://docs.aws.amazon.com/AWSDynamoDB/latest/APIReference/API_BatchExecuteStatement.html) in *AWS SDK for JavaScript API Reference*.

### Scan a table

The following code example shows how to scan a DynamoDB table.

**SDK for JavaScript (v3)**

```javascript
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' },
        ...,
    });

    const response = await docClient.send(command);
    console.log(response);
    return response;
};
```
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.
    // 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",
};
Update an item in a table

The following code example shows how to update an item in a DynamoDB table.

**SDK for JavaScript (v3)**

For more information, see [AWS SDK for JavaScript Developer Guide](#).

For API details, see [Scan](#) 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](#).

This example uses the document client to simplify working with items in DynamoDB. For API details see [UpdateCommand](#).

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

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);
```
```javascript
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](https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWSDynamoDB/ApiReference/API_UpdateItem.html) in AWS SDK for JavaScript API Reference.

### Write a batch of items

The following code example shows how to write a batch of DynamoDB items.

**SDK for JavaScript (v3)**

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

This example uses the document client to simplify working with items in DynamoDB. For API details see [BatchWrite](https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWSDynamoDB/ApiReference/API_BatchWrite.html).
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-sdk-examples/libs/utils/util-fs.js";
import { chunkArray } from "@aws-sdk-examples/libs/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: {
        "BatchWriteMoviesTable": putRequests,
      },
    });

    await docClient.send(command);
  }
};
For API details, see [BatchWriteItem](https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/dynamodb-api-reference.html#batchwriteitem) in *AWS SDK for JavaScript API Reference*.

**SDK for JavaScript (v2)**

```javascript
// 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" }
          }
        },
      }
    ]
  }
};
```
Scenarios

Get started with tables, items, and queries

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.

```javascript
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 BOOL) 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-sdk-examples/libs/utils/util-string.js";
import { dirnameFromMetaUrl } from "@aws-sdk-examples/libs/utils/util-fs.js";
import { chunkArray } from "@aws-sdk-examples/libs/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,
    }) from "@aws-sdk/client-dynamodb";

}
// 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
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)}`);
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/
  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**

### 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](https://github.com/aws-samples/aws-sdk-js-code-examples).

Execute batch PartiQL statements.
import {
    BillingMode,
    CreateTableCommand,
    DeleteTableCommand,
    DynamoDBClient,
    waitUntilTableExists,
} from "@aws-sdk/client-dynamodb";
import {
    DynamoDBDocumentClient,
    BatchExecuteStatementCommand,
} from "@aws-sdk/lib-dynamodb";

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

const log = (msg) => console.log(`[SCENARIO] ${msg}`);
const tableName = "Cities";

export const main = async () => {
    /**
     * 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

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.");

// The base client is used here instead of 'lib-dynamodb'. There's
// causing list parameters to not be handled correctly.
const addItemsStatementCommand = new BatchExecuteStatementCommand({
    Statements: [
        {
            Statement: 'INSERT INTO ${tableName} value {name:?, population:?}',
            Parameters: ['Alachua', 10712],
        },
        {
            Statement: 'INSERT INTO ${tableName} value {name:?, population:?}',
            Parameters: ['High Springs', 6415],
        },
    ],
});

await client.send(addItemsStatementCommand);

log('Cities inserted.');

/**
* Select items.

```
log("Selecting cities from the table.");
const selectItemsStatementCommand = new BatchExecuteStatementCommand({
  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.");
// The base client is used here for the same reasons described previously.
const updateItemStatementCommand = new BatchExecuteStatementCommand({
  Statements: [
    {
      Statement: `UPDATE ${tableName} SET population=? WHERE name=?`,
      Parameters: [10, "Alachua"],
    },
    {
      Statement: `UPDATE ${tableName} SET population=? WHERE name=?`,
      Parameters: [5, "High Springs"],
    },
  ],
});
```
await client.send(updateItemStatementCommand);
log(`Updated cities.`);

/**
 * Delete the items.
 */

log("Deleting the cities.");
const deleteItemStatementCommand = new BatchExecuteStatementCommand({
  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](https://github.com/aws-samples/aws-sdk-for-javascript).

Execute single PartiQL statements.

```javascript
import {
  BillingMode,
  CreateTableCommand,
  DeleteTableCommand,
  DynamoDBClient,
  waitUntilTableExists,
  ExecuteStatementCommand as BaseExecuteStatementCommand,
} from '@aws-sdk/client-dynamodb';
import {
  DynamoDBDocumentClient,
  ExecuteStatementCommand,
} from '@aws-sdk/lib-dynamodb';

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);
const log = (msg) => console.log(`[SCENARIO] ${msg}`);
const tableName = "SingleOriginCoffees";

export const main = async () => {
  /**
   * 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.");
// The base client is used here instead of 'lib-dynamodb'. There's
// causing list parameters to not be handled correctly.
```javascript
const addItemStatementCommand = new BaseExecuteStatementCommand({
    Statement: `INSERT INTO ${tableName} value {'varietal':?, 'profile':?}`,
    Parameters: [
        { S: "arabica" },
        { L: [{ S: "chocolate" }, { S: "floral" }] },
    ],
});
await client.send(addItemStatementCommand);
log('Coffee inserted.~');

/**
 * Select an item.
 */

log("Selecting the coffee from the table.");
const selectItemStatementCommand = new ExecuteStatementCommand({
    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.");
// The base client is used here for the same reasons described previously.
const updateItemStatementCommand = new BaseExecuteStatementCommand({
    Statement: `UPDATE ${tableName} SET profile=list_append(profile, ?) WHERE varietal=?`,
    Parameters: [[ L: [{ S: "fruity" } }, { S: "arabica" }],
    ],
});
await client.send(updateItemStatementCommand);
log(`Updated coffee`);

/**
 * Delete the item.
 */
```
```javascript
/*

log("Deleting the coffee.");
const deleteItemStatementCommand = new ExecuteStatementCommand({
    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](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ql-reference.delete.html) in *AWS SDK for JavaScript API Reference*.

## 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.

*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 and cross-service examples.

*Scenarios* are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.

Each example includes a link to GitHub, 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)

```javascript
import { DescribeSecurityGroupsCommand } from '@aws-sdk/client-ec2';

import { client } from './libs/client.js';

// Call DescribeSecurityGroups and display the result.
export const main = async () => {
  try {
    const { SecurityGroups } = await client.send(
      new DescribeSecurityGroupsCommand({}),
    );

    const securityGroupList = SecurityGroups.slice(0, 9)
      .map((sg) => ` • ${sg.GroupId}: ${sg.GroupName}`)
      .join('
');

    console.log(
      "Hello, Amazon EC2! Let's list up to 10 of your security groups:",
    );
    console.log(securityGroupList);
  } catch (err) {
    console.error(err);
  }
};
```

- For API details, see `DescribeSecurityGroups` in AWS SDK for JavaScript API Reference.
Actions

Allocate an Elastic IP address

The following code example shows how to allocate an Elastic IP address for Amazon EC2.

SDK for JavaScript (v3)

```javascript
import { AllocateAddressCommand } from '@aws-sdk/client-ec2';

import { client } from '../libs/client.js';

export const main = async () => {
  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 (err) {
    console.error(err);
  }
};
```

For API details, see AllocateAddress in AWS SDK for JavaScript API Reference.
## Associate an Elastic IP address with an instance

The following code example shows how to associate an Elastic IP address with an Amazon EC2 instance.

### SDK for JavaScript (v3)

```javascript
import { AssociateAddressCommand } from '@aws-sdk/client-ec2';
import { client } from '../libs/client.js';

export const main = async () => {
  // You need to allocate an Elastic IP address before associating it with an instance.
  // You can do that with the AllocateAddressCommand.
  const allocationId = 'ALLOCATION_ID';
  // You need to create an EC2 instance before an IP address can be associated with it.
  // You can do that with the RunInstancesCommand.
  const instanceId = 'INSTANCE_ID';
  const command = new AssociateAddressCommand({
    AllocationId: allocationId,
    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 (err) {
    console.error(err);
  }
};
```

---

**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws/samples).
• For API details, see AssociateAddress in AWS SDK for JavaScript API Reference.

Create a launch template

The following code example shows how to create an Amazon EC2 launch template.

SDK for JavaScript (v3)

```javascript
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.
Create a security group

The following code example shows how to create an Amazon EC2 security group.

SDK for JavaScript (v3)

```javascript
import { CreateSecurityGroupCommand } from '@aws-sdk/client-ec2';
import { client } from '../libs/client.js';

export const main = async () => {
  const command = new CreateSecurityGroupCommand(
    // Up to 255 characters in length. Cannot start with sg-.
    { GroupName: "SECURITY_GROUP_NAME", Description: "DESCRIPTION" },
  );

  try {
    const { GroupId } = await client.send(command);
    console.log(GroupId);
  } catch (err) {
    console.error(err);
  }
};
```

- For API details, see [CreateSecurityGroup](https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/EC2.html#CreateSecurityGroupCommand) in [AWS SDK for JavaScript API Reference](https).

Create a security key pair

The following code example shows how to create a security key pair for Amazon EC2.
import { CreateKeyPairCommand } from '@aws-sdk/client-ec2';

import { client } from '../libs/client.js';

export const main = async () => {
try {
  // Create a key pair in Amazon EC2.
  const { KeyMaterial, KeyName } = await client.send(
    // A unique name for the key pair. Up to 255 ASCII characters.
    new CreateKeyPairCommand({ KeyName: 'KEY_PAIR_NAME' }),
  );
  // This logs your private key. Be sure to save it.
  console.log(KeyName);
  console.log(KeyMaterial);
} catch (err) {
  console.error(err);
}
};

• For API details, see CreateKeyPair in AWS SDK for JavaScript API Reference.

Create and run an instance

The following code example shows how to create and run an Amazon EC2 instance.

SDK for JavaScript (v3)
import { RunInstancesCommand } from "@aws-sdk/client-ec2";

import { client } from "./libs/client.js";

// Create a new EC2 instance.
export const main = async () => {
    const command = new RunInstancesCommand({
        // Your key pair name.
        KeyName: "KEY_PAIR_NAME",
        // Your security group.
        SecurityGroupIds: ["SECURITY_GROUP_ID"],
        // An x86_64 compatible image.
        ImageId: "ami-0001a0d1a04bfcc30",
        // An x86_64 compatible free-tier instance type.
        InstanceType: "t1.micro",
        // Ensure only 1 instance launches.
        MinCount: 1,
        MaxCount: 1,
    });

    try {
        const response = await client.send(command);
        console.log(response);
    } catch (err) {
        console.error(err);
    }
};

• For API details, see RunInstances in AWS SDK for JavaScript API Reference.

Delete a launch template

The following code example shows how to delete an Amazon EC2 launch template.

SDK for JavaScript (v3)

ℹ️ Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.
```javascript
await client.send(
    new DeleteLaunchTemplateCommand({
        LaunchTemplateName: NAMES.launchTemplateName,
    })),
);
```

- For API details, see [DeleteLaunchTemplate](#) in *AWS SDK for JavaScript API Reference*.

### Delete a security group

The following code example shows how to delete an Amazon EC2 security group.

#### SDK for JavaScript (v3)

```javascript
import { DeleteSecurityGroupCommand } from '@aws-sdk/client-ec2';
import { client } from '../libs/client.js';

export const main = async () => {
    const command = new DeleteSecurityGroupCommand({
        GroupId: "GROUP_ID",
    });

    try {
        await client.send(command);
        console.log("Security group deleted successfully.");
    } catch (err) {
        console.error(err);
    }
};
```

- For API details, see [DeleteSecurityGroup](#) in *AWS SDK for JavaScript API Reference*.
Delete a security key pair

The following code example shows how to delete an Amazon EC2 security key pair.

SDK for JavaScript (v3)

```javascript
import { DeleteKeyPairCommand } from '@aws-sdk/client-ec2';
import { client } from '../libs/client.js';

export const main = async () => {
  const command = new DeleteKeyPairCommand({
    KeyName: "KEYPAIR_NAME",
  });
  try {
    await client.send(command);
    console.log("Successfully deleted key pair.");
  } catch (err) {
    console.error(err);
  }
};
```

- For API details, see [DeleteKeyPair](#) in AWS SDK for JavaScript API Reference.

Describe Regions

The following code example shows how to describe Amazon EC2 Regions.

import { DescribeRegionsCommand } from '@aws-sdk/client-ec2';

import { client } from '../libs/client.js';

export const main = async () => {
  const command = new DescribeRegionsCommand({
    // By default this command will not show regions that require you to opt-in.
    // When AllRegions true even the regions that require opt-in will be returned.
    AllRegions: true,
    // You can omit the Filters property if you want to get all regions.
    Filters: [
      {
        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: ['ap-southeast-4'],
      },
    ],
  });

  try {
    const { Regions } = await client.send(command);
    const regionsList = Regions.map((reg) => ` • ${reg.RegionName}`);
    console.log("Found regions:");
    console.log(regionsList.join("\n"));
  } catch (err) {
    console.error(err);
  }
};

For API details, see DescribeRegions in AWS SDK for JavaScript API Reference.
Describe instances

The following code example shows how to describe Amazon EC2 instances.

SDK for JavaScript (v3)

```javascript
import { DescribeInstancesCommand } from '@aws-sdk/client-ec2';
import { client } from '../libs/client.js';

// List all of your EC2 instances running with x86_64 architecture that were
// launched this month.
export const main = async () => {
  const d = new Date();
  const year = d.getFullYear();
  const month = `0${d.getMonth() + 1}`.slice(-2);
  const launchTimePattern = `${year}-${month}-*`;
  const command = new DescribeInstancesCommand({
    Filters: [
      { Name: "architecture", Values: ["x86_64"] },
      { Name: "instance-state-name", Values: ["running"] },
      { Name: "launch-time",
        Values: [launchTimePattern],
      },
    ],
  });

  try {
    const { Reservations } = await client.send(command);
    const instanceList = Reservations.reduce((prev, current) => {
      return prev.concat(current.Instances);
    }, []);

    console.log(instanceList);
  } catch (err) {
```

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://aws.amazon.com/code-examples/).
For API details, see [DescribeInstances](#) in [AWS SDK for JavaScript API Reference](#).

### Disable detailed monitoring

The following code example shows how to disable detailed monitoring on an Amazon EC2 instance.

#### SDK for JavaScript (v3)

```javascript
import { UnmonitorInstancesCommand } from '@aws-sdk/client-ec2';
import { client } from '../libs/client.js';

export const main = async () => {
    const command = new UnmonitorInstancesCommand({
        InstanceIds: ['i-09a3dfe7ae00e853f'],
    });

    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('
'));
    } catch (err) {
    console.error(err);
    }
};
```

---

*Note*

There’s more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).
Disassociate an Elastic IP address from an instance

The following code example shows how to disassociate an Elastic IP address from an Amazon EC2 instance.

SDK for JavaScript (v3)

```javascript
import { DisassociateAddressCommand } from '@aws-sdk/client-ec2';
import { client } from '../libs/client.js';

// Disassociate an Elastic IP address from an instance.
export const main = async () => {
  const command = new DisassociateAddressCommand({
    // You can also use PublicIp, but that is for EC2 classic which is being retired.
    AssociationId: "ASSOCIATION_ID",
  });

  try {
    await client.send(command);
    console.log("Successfully disassociated address");
  } catch (err) {
    console.error(err);
  }
};
```

• For API details, see [DisassociateAddress](#) in *AWS SDK for JavaScript API Reference.*
Enable monitoring

The following code example shows how to enable monitoring for a running Amazon EC2 instance.

SDK for JavaScript (v3)

```javascript
import { MonitorInstancesCommand } from '@aws-sdk/client-ec2';

import { client } from '../libs/client.js';

// 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.
export const main = async () => {
  const command = new MonitorInstancesCommand({
    InstanceIds: ['INSTANCE_ID'],
  });

  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 (err) {
    console.error(err);
  }
};
```

- For API details, see MonitorInstances in AWS SDK for JavaScript API Reference.
Get data about Amazon Machine Images

The following code example shows how to get data about Amazon Machine Images (AMIs).

SDK for JavaScript (v3)

```javascript
import { paginateDescribeImages } from '@aws-sdk/client-ec2';

import { client } from './libs/client.js';

// List at least the first i386 image available for EC2 instances.
export const main = async () => {
  // 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: 25 },
    {
      // 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: ["x86_64"] }],
    },
  );

  try {
    const arm64Images = [];
    for await (const page of paginator) {
      if (page.Images.length) {
        arm64Images.push(...page.Images);
        // Once we have at least 1 result, we can stop.
        if (arm64Images.length >= 1) {
          break;
        }
      }
    }
  }
```

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 data about a security group

The following code example shows how to get data about an Amazon EC2 security group.

**SDK for JavaScript (v3)**

```javascript
import { DescribeSecurityGroupsCommand } from '@aws-sdk/client-ec2';
import { client } from '../libs/client.js';

// Log the details of a specific security group.
export const main = async () => {
  const command = new DescribeSecurityGroupsCommand({
    GroupIds: ['SECURITY_GROUP_ID'],
  });
  try {
    const { SecurityGroups } = await client.send(command);
    console.log(JSON.stringify(SecurityGroups, null, 2));
  } catch (err) {
    console.error(err);
  }
};
```

- For API details, see [DescribeImages](https://aws.amazon.com) in AWS SDK for JavaScript API Reference.
For API details, see `DescribeSecurityGroups` in *AWS SDK for JavaScript API Reference*.

Get data about instance types

The following code example shows how to get data about Amazon EC2 instance types.

SDK for JavaScript (v3)

```javascript
import {
  paginateDescribeInstanceTypes,
  DescribeInstanceTypesCommand,
} from '@aws-sdk/client-ec2';

import { client } from '../libs/client.js';

// List at least the first arm64 EC2 instance type available.
export const main = async () => {
  // 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: 25 },
    {
      Filters: [
        { Name: 'processor-info.supported-architecture', Values: ['x86_64'] },
        { Name: 'free-tier-eligible', Values: ['true'] },
      ],
    }
  );

  try {
    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(instanceTypes);
} catch (err) {
    console.error(err);
}
Get details about Elastic IP addresses

The following code example shows how to get details about Elastic IP addresses.

**SDK for JavaScript (v3)**

```javascript
import { DescribeAddressesCommand } from "@aws-sdk/client-ec2";

import { client } from "./libs/client.js";

export const main = async () => {
  const command = new DescribeAddressesCommand({
    // You can omit this property to show all addresses.
    AllocationIds: ["ALLOCATION_ID"],
  });

  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 (err) {
    console.error(err);
  }
};
```

For API details, see **DescribeAddresses** in *AWS SDK for JavaScript API Reference*.
Get the default VPC

The following code example shows how to get the default VPC of the current account.

SDK for JavaScript (v3)

```javascript
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](#).

Get the default subnets for a VPC

The following code example shows how to get the default subnets for a VPC.

SDK for JavaScript (v3)

```javascript
const client = new EC2Client({});
const { Subnets } = await client.send(
  new DescribeSubnetsCommand({
    Filters: [
      { Name: "vpc-id", Values: [state.defaultVpc] },
    ],
  }));
```

- For API details, see [DescribeSubnets](#) in [AWS SDK for JavaScript API Reference](#).
• For API details, see DescribeSubnets in AWS SDK for JavaScript API Reference.

List security key pairs

The following code example shows how to list Amazon EC2 security key pairs.

SDK for JavaScript (v3)

```javascript
import { DescribeKeyPairsCommand } from '@aws-sdk/client-ec2';
import { client } from '../libs/client.js';

export const main = async () => {
    const command = new DescribeKeyPairsCommand({});
    try {
        const { KeyPairs } = await client.send(command);
        const keyPairList = KeyPairs.map((kp) => ` • ${kp.KeyPairId}: ${kp.KeyName}`).join('
');
        console.log('The following key pairs were found in your account:');
        console.log(keyPairList);
    } catch (err) {
        console.error(err);
    }
}
```

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 API details, see `DescribeKeyPairs` in `AWS SDK for JavaScript API Reference`.

Reboot an instance

The following code example shows how to reboot an Amazon EC2 instance.

**SDK for JavaScript (v3)**

```javascript
import { RebootInstancesCommand } from '@aws-sdk/client-ec2';
import { client } from '../libs/client.js';

export const main = async () => {
    const command = new RebootInstancesCommand({
        InstanceIds: ['INSTANCE_ID'],
    });

    try {
        await client.send(command);
        console.log('Instance rebooted successfully.');
    } catch (err) {
        console.error(err);
    }
};
```

• For API details, see `RebootInstances` in `AWS SDK for JavaScript API Reference`.

Release an Elastic IP address

The following code example shows how to release an Elastic IP address.
import { ReleaseAddressCommand } from "@aws-sdk/client-ec2";

import { client } from "./libs/client.js";

export const main = async () => {
  const command = new ReleaseAddressCommand({
    // You can also use PublicIp, but that is for EC2 classic which is being retired.
    AllocationId: "ALLOCATION_ID",
  });

  try {
    await client.send(command);
    console.log("Successfully released address.");
  } catch (err) {
    console.error(err);
  }
};

• For API details, see ReleaseAddress in AWS SDK for JavaScript API Reference.

Replace the instance profile associated with an instance

The following code example shows how to replace the instance profile associated with an Amazon EC2 instance.
SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```javascript
await retry({ intervalInMs: 1000, maxRetries: 30 }, () =>
  ec2Client.send(
    new ReplaceIamInstanceProfileAssociationCommand({
      AssociationId: state.instanceProfileAssociationId,
      IamInstanceProfile: { Name: NAMES.ssmOnlyInstanceProfileName },
    }),
  ),
);
```

- For API details, see [ReplaceIamInstanceProfileAssociation](https://aws-sdk.github.io/aws-sdk-js-v3/api-docs/replaceiaminstanceprofileassociation.html) in AWS SDK for JavaScript API Reference.

Set inbound rules for a security group

The following code example shows how to set inbound rules for an Amazon EC2 security group.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```javascript
import { AuthorizeSecurityGroupIngressCommand } from '@aws-sdk/client-ec2';
import { client } from '../libs/client.js';

// Grant permissions for a single IP address to ssh into instances
// within the provided security group.
```
export const main = async () => {
  const command = new AuthorizeSecurityGroupIngressCommand({
    // Replace with a security group ID from the AWS console or
    // the DescribeSecurityGroupsCommand.
    GroupId: "SECURITY_GROUP_ID",
    IpPermissions: [
      {
        IpProtocol: "tcp",
        FromPort: 22,
        ToPort: 22,
        // Replace 0.0.0.0 with 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: "0.0.0.0/32" }],
      },
    ],
  });

  try {
    const { SecurityGroupRules } = await client.send(command);
    console.log(JSON.stringify(SecurityGroupRules, null, 2));
  } catch (err) {
      console.error(err);
  }
};

• For API details, see AuthorizeSecurityGroupIngress in AWS SDK for JavaScript API Reference.

Start an instance

The following code example shows how to start an Amazon EC2 instance.

SDK for JavaScript (v3)

⚠️ 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 { StartInstancesCommand } from "@aws-sdk/client-ec2";

import { client } from "./libs/client.js";

export const main = async () => {
  const command = new StartInstancesCommand({
    // Use DescribeInstancesCommand to find InstanceIds
    InstanceIds: ["INSTANCE_ID"],
  });

  try {
    const { StartingInstances } = await client.send(command);
    const instanceIdList = StartingInstances.map(
      (instance) => ` • ${instance.InstanceId}`,
    );
    console.log("Starting instances:");
    console.log(instanceIdList.join("\n"));
  } catch (err) {
      console.error(err);
  }
};

- For API details, see StartInstances in AWS SDK for JavaScript API Reference.

Stop an instance

The following code example shows how to stop an Amazon EC2 instance.

SDK for JavaScript (v3)

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 { StopInstancesCommand } from "@aws-sdk/client-ec2";

import { client } from "./libs/client.js";
export const main = async () => {
  const command = new StopInstancesCommand({
    // Use DescribeInstancesCommand to find InstanceIds
    InstanceIds: ["INSTANCE_ID"],
  });

  try {
    const { StoppingInstances } = await client.send(command);
    const instanceIdList = StoppingInstances.map(
      (instance) => ` • ${instance.InstanceId}`,
    );
    console.log("Stopping instances:");
    console.log(instanceIdList.join("\n"));
  } catch (err) {
    console.error(err);
  }
};

• For API details, see StopInstances in AWS SDK for JavaScript API Reference.

Terminate an instance

The following code example shows how to terminate an Amazon EC2 instance.

SDK for JavaScript (v3)

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 { TerminateInstancesCommand } from "@aws-sdk/client-ec2";
import { client } from "./libs/client.js";

export const main = async () => {
  const command = new TerminateInstancesCommand({
    InstanceIds: ["INSTANCE_ID"],
try {
  const { TerminatingInstances } = await client.send(command);
  const instanceList = TerminatingInstances.map(
    (instance) => ` • ${instance.InstanceId}`
  );
  console.log("Terminating instances:");
  console.log(instanceList.join("\n"));
} catch (err) {
  console.error(err);
}

• For API details, see TerminatingInstances 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.
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-sdk-examples/libs/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.

```javascript
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.

```javascript
/*
 * Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
 * SPDX-License-Identifier: Apache-2.0
 */
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,
} from "@aws-sdk/client-ec2";
```
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-sdk-examples/libs/scenario/index.js";
import { retry } from "@aws-sdk-examples/libs/utils/util-timers.js";

import { MESSAGES, NAMES, RESOURCES_PATH, ROOT } from "./constants.js";
import { initParamsSteps } from "./steps-reset-params.js";

/**
 * @type {import('@aws-sdk-examples/libs/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("${KEY_PAIR_NAME}", NAMES.keyPairName),
   ),
});
new ScenarioOutput("creatingInstancePolicy", async (state) => {
    const client = new IAMClient({});
    const {
        Policy: { Arn },
    } = await client.send(new CreatePolicyCommand({
        PolicyName: NAMES.instancePolicyName,
        PolicyDocument: readFileSync(join(RESOURCES_PATH, "instance_policy.json"),
    }));
    state.instancePolicyArn = Arn;
}),
new ScenarioOutput("createdInstancePolicy", (state) =>
    MESSAGES.createdInstancePolicy
    .replace("${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("${INSTANCE_PROFILE_NAME}", NAMES.instanceProfileName)
    .replace("${INSTANCEPROFILE_ARN}", state.instanceProfileArn),
),
new ScenarioOutput("addingRoleToInstanceProfile",
    MESSAGES.addingRoleToProfile
    .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.addedRoleToProfile
    .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,
        },
    })
);  
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: {
                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: {
                    InstanceType: "t3.micro",
                    ImageId: Parameter.Value,
                    IamInstanceProfile: { Name: NAMES.instanceProfileName },
                    UserData: readFileSync(
                        join(RESOURCES_PATH, "server_startup_script.sh"),
                    ).toString("base64"),
                    KeyName: NAMES.keyPairName,
                },
            })
        );
});
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("${AVAILABILITY_ZONE_NAMES}",
      state.availabilityZoneNames.join(" ",
    ),
),
new ScenarioInput("confirmContinue", MESSAGES.confirmContinue, {
  type: "confirm",
}),
new ScenarioOutput("loadBalancer", MESSAGES.loadBalancer),
new ScenarioOutput("gettingVpc", MESSAGES.gettingVpc),
new ScenarioAction("getVpc", async (state) => {
  // snippet-start:[javascript.v3.wkflw.resilient.DescribeVpcs
  const client = new EC2Client({});
  const { Vpcs } = await client.send(
    new DescribeVpcsCommand({
      Filters: [{ Name: "is-default", Values: ["true"] }],
    }),
  );
  // snippet-end:[javascript.v3.wkflw.resilient.DescribeVpcs
  state.defaultVpc = Vpcs[0].VpcId;
}),
new ScenarioOutput("gotVpc", (state) =>
  MESSAGES.gotVpc.replace("${VPC_ID}", state.defaultVpc),
),
new ScenarioOutput("gettingSubnets", MESSAGES.gettingSubnets),
new ScenarioAction("getSubnets", async (state) => {
  // snippet-start:[javascript.v3.wkflw.resilient.DescribeSubnets

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"] },
        ],
    }));
// snippet-end:[javascript.v3.wkflw.resilient.DescribeSubnets]
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) => {
    // snippet-start:[javascript.v3.wkflw.resilient.CreateTargetGroup]
    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) => {
  // snippet-start:[javascript.v3.wkflw.resilient.CreateLoadBalancer]
  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] },
  );
  // snippet-end:[javascript.v3.wkflw.resilient.CreateLoadBalancer]
  
  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) => {
  // snippet-start:[javascript.v3.wkflw.resilient.CreateListener]
  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 },
      ],
    }),
  );
  // snippet-end:[javascript.v3.wkflw.resilient.CreateListener]
  const listener = Listeners[0];
  state.loadBalancerListenerArn = listener.ListenerArn;

rfn  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) => {
  // snippet-start:[javascript.v3.wkflw.resilient.AttachTargetGroup]
  const client = new AutoScalingClient({});
  await client.send(
    new AttachLoadBalancerTargetGroupsCommand({
      AutoScalingGroupName: NAMES.autoScalingGroupName,
      TargetGroupARNs: [state.targetGroupArn],
    }),
  );
  // snippet-end:[javascript.v3.wkflw.resilient.AttachTargetGroup]
}),
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",
      }),
    );
  },
Create steps to run the demo.

```javascript
/*
 * Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
 * SPDX-License-Identifier: Apache-2.0
 */
```
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-sdk-examples/libs/scenario/scenario.js";
import { retry } from "@aws-sdk-examples/libs/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:
${JSON.stringify(state.recommendation, null, 2)}`,
        { preformatted: true },
    });

const getHealthCheck = new ScenarioAction("getHealthCheck", async (state) => {
    // snippet-start:[javascript.v3.wkflw.resilient.DescribeTargetGroups]
    const client = new ElasticLoadBalancingV2Client({});
    const { TargetGroups } = await client.send(
        new DescribeTargetGroupsCommand({
            Names: [NAMES.loadBalancerTargetGroupName],
        })),
    );
    // snippet-end:[javascript.v3.wkflw.resilient.DescribeTargetGroups]

    // snippet-start:[javascript.v3.wkflw.resilient.DescribeTargetHealth]
    const { TargetHealthDescriptions } = await client.send(
        new DescribeTargetHealthCommand({
            TargetGroupArn: TargetGroups[0].TargetGroupArn,
        })),
    );
    // snippet-end:[javascript.v3.wkflw.resilient.DescribeTargetHealth]
state.targetHealthDescriptions = TargetHealthDescriptions;
});

const getHealthCheckResult = new ScenarioOutput(
  "getHealthCheckResult",
  /**
   * @param {{ targetHealthDescriptions: import('@aws-sdk/client-elastic-load-
   * balacing-v2').TargetHealthDescription[]} state
   */
   (state) => {
     const status = state.targetHealthDescriptions
       .map((th) => `${th.Target.Id}: ${th.TargetHealth.State}`)
       .join("\n");
     return `Health check:
${status}`;
   },
   { preformatted: true },
);

const loadBalancerLoop = new ScenarioAction(
  "loadBalancerLoop",
  getRecommendation.action,
  {
    whileConfig: {
      inputEquals: true,
      input: new ScenarioInput(
        "loadBalancerCheck",
        MESSAGES.demoLoadBalancerCheck,
        {
          type: "confirm",
        },
        output: getRecommendationResult,
      },
    },
  },
);

const healthCheckLoop = new ScenarioAction(
  "healthCheckLoop",
  getHealthCheck.action,
  {
    whileConfig: {
      inputEquals: true,
    },
  },
);
input: new ScenarioInput("healthCheck", MESSAGES.demoHealthCheck, {
    type: "confirm",
}),
output: getHealthCheckResult,
}],
);

const statusSteps = [
    getRecommendation,
    getRecommendationResult,
    getHealthCheck,
    getHealthCheckResult,
];

/**
 * @type {import('@aws-sdk-examples/libs/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,
    })
  );
});
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];
    // snippet-start:
    [javascript.v3.wkflw.resilient.DescribeIamInstanceProfileAssociations]
    const ec2Client = new EC2Client({});
    const { IamInstanceProfileAssociations } = await ec2Client.send(
      new DescribeIamInstanceProfileAssociationsCommand({
        Filters: [
          { Name: "instance-id", Values: [state.targetInstance.InstanceId] },
        ],
      }));
    // snippet-end:
    [javascript.v3.wkflw.resilient.DescribeIamInstanceProfileAssociations]
    state.instanceProfileAssociationId = 
      IamInstanceProfileAssociations[0].AssociationId;
    // snippet-start:
    [javascript.v3.wkflw.resilient.ReplaceIamInstanceProfileAssociation]
    await retry({ intervalInMs: 1000, maxRetries: 30 }, () =>
      ec2Client.send(
        new ReplaceIamInstanceProfileAssociationCommand({
          AssociationId: state.instanceProfileAssociationId,
          IamInstanceProfile: { Name: NAMES.ssmOnlyInstanceProfileName },
        }));
    // snippet-end:
    [javascript.v3.wkflw.resilient.ReplaceIamInstanceProfileAssociation]
```javascript
// snippet-end:
[javascript.v3.wkflw.resilient.ReplaceIamInstanceProfileAssociation]

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,
    ),
```
Amazon EC2

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,
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",
                    },
                ],
            }),
        }),
    );
}
Create steps to destroy all of the resources.

/*
 * Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
 * SPDX-License-Identifier: Apache-2.0
*/
import { unlinkSync } from "node:fs";

import { DynamoDBClient, DeleteTableCommand } from "@aws-sdk/client-dynamodb";
import {
    EC2Client,
    DeleteKeyPairCommand,
    DeleteLaunchTemplateCommand,
} 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-sdk-examples/libs/scenario/index.js";
import { retry } from "@aws-sdk-examples/libs/utils/util-timers.js";

import { MESSAGES, NAMES } from "./constants.js";
import { findLoadBalancer } from "./shared.js";

/**
 * @type {import('@aws-sdk-examples/libs/scenario.js').Step[]}
export const destroySteps = [
  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("${TABLE_NAME}Pair", NAMES.keyPairName);
    }
  })
]
} else {
    return MESSAGES.deletedKeyPair.replace(
        "${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 ${NAMES.instancePolicyName} not found.`,
            );
        } else {
            await client.send(
                new DetachRolePolicyCommand({
                    RoleName: NAMES.instanceRoleName,
                    PolicyArn: policy.Arn,
                }),
            );
        }
    } catch (e) {
        state.detachPolicyFromRoleError = e;
    }
}),
new ScenarioOutput("detachedPolicyFromRole", (state) => {
    if (state.detachPolicyFromRoleError) {
        console.error(state.detachPolicyFromRoleError);
        return MESSAGES.detachedPolicyFromRoleError
            .replace("${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)
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 {
    // snippet-start:[javascript.v3.wkflw.resilient.DeleteInstanceProfile]
    const client = new IAMClient({});
    await client.send(
      new DeleteInstanceProfileCommand({
        InstanceProfileName: NAMES.instanceProfileName,
      }),
    );
    // snippet-end:[javascript.v3.wkflw.resilient.DeleteInstanceProfile]
  } 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("deleteLaunchTemplate", async (state) => {
  const client = new EC2Client({});
  try {
    // snippet-start:[javascript.v3.wkflw.resilient.DeleteLaunchTemplate]
    await client.send(
      new DeleteLaunchTemplateCommand({
        LaunchTemplateName: NAMES.launchTemplateName,
      }),
    );
    // snippet-end:[javascript.v3.wkflw.resilient.DeleteLaunchTemplate]
  } 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("deleteAutoScalingGroup", async (state) => {
    try {
        await terminateGroupInstances(NAMES.autoScalingGroupName);
        await retry({ intervalInMs: 30000, 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("deleteLoadBalancer", async (state) => {
    try {
        // snippet-start:[javascript.v3.wkflw.resilient.DeleteLoadBalancer]
        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.");
            }
        });
        // snippet-end:[javascript.v3.wkflw.resilient.DeleteLoadBalancer]
    } 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) => {
    try {
        const client = new ElasticLoadBalancingV2Client({});
        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;
    }
    // snippet-end:[javascript.v3.wkflw.resilient.DeleteTargetGroup]
}),

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,
        );
    }
})
);  
} 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,
    );
  }
});
"${POLICY_NAME}",
   NAMES.ssmOnlyPolicyName,
 );
}
});
new ScenarioAction("deleteSsmOnlyRole", async (state) => {
  try {
    const iamClient = new IAMClient({});
    await iamClient.send(
      new DeleteRoleCommand(
        {
          RoleName: NAMES.ssmOnlyRoleName,
        }
      ),
    );
  } catch (e) {
    state.deleteSsmOnlyRoleError = e;
  }
});
new ScenarioOutput("deleteSsmOnlyRoleResult", (state) => {
  if (state.deleteSsmOnlyRoleError) {
    console.error(state.deleteSsmOnlyRoleError);
    return MESSAGES.deleteSsmOnlyRoleError.replace(
      "${ROLE_NAME}",
      NAMES.ssmOnlyRoleName,
    );
  } else {
    return MESSAGES.deletedSsmOnlyRole.replace(
      "${ROLE_NAME}",
      NAMES.ssmOnlyRoleName,
    );
  }
});
];

/**
 * @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;
    }
}
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;
        }
    }
}

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**
• **ReplaceIamInstanceProfileAssociation**
• **TerminateInstanceInAutoScalingGroup**
• **UpdateAutoScalingGroup**

### Get started with instances

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](https://github.com/aws-examples). Run an interactive scenario at a command prompt.

```javascript
import { mkdtempSync, writeFileSync, rmSync } from "fs";
import { tmpdir } from "os";
import { join } from "path";
import { get } from "http";
```
AllocateAddressCommand,
AssociateAddressCommand,
AuthorizeSecurityGroupIngressCommand,
CreateKeyPairCommand,
CreateSecurityGroupCommand,
DeleteKeyPairCommand,
DeleteSecurityGroupCommand,
DescribeInstancesCommand,
DescribeKeyPairsCommand,
DescribeSecurityGroupsCommand,
DisassociateAddressCommand,
EC2Client,
paginateDescribeImages,
paginateDescribeInstanceTypes,
ReleaseAddressCommand,
RunInstancesCommand,
StartInstancesCommand,
StopInstancesCommand,
TerminateInstancesCommand,
waitUntilInstanceStatusOk,
waitUntilInstanceStopped,
waitUntilInstanceTerminated,
} from "@aws-sdk/client-ec2";
import { paginateGetParametersByPath, SSMClient } from "@aws-sdk/client-ssm";
import {
  promptToSelect,
promptToContinue,
} from "@aws-sdk-examples/libs/utils/util-io.js";
import { wrapText } from "@aws-sdk-examples/libs/utils/util-string.js";

const ec2Client = new EC2Client();
const ssmClient = new SSMClient();

const tmpDirectory = mkdtempSync(join(tmpdir(), "ec2-scenario-tmp"));

const createKeyPair = async (keyPairName) => {
  // Create a key pair in Amazon EC2.
  const { KeyMaterial, KeyPairId } = await ec2Client.send(
    // A unique name for the key pair. Up to 255 ASCII characters.
    new CreateKeyPairCommand({ KeyName: keyPairName });
  );

  // Save the private key in a temporary location.
writeFileSync(`${tmpDirectory}/${keyPairName}.pem`, KeyMaterial, {
    mode: 0o400,
});

return KeyPairId;
};

const describeKeyPair = async (keyPairName) => {
    const command = new DescribeKeyPairsCommand({
        KeyNames: [keyPairName],
    });
    const { KeyPairs } = await ec2Client.send(command);
    return KeyPairs[0];
};

const createSecurityGroup = async (securityGroupName) => {
    const command = new CreateSecurityGroupCommand({
        GroupName: securityGroupName,
        Description: "A security group for the Amazon EC2 example.",
    });
    const { GroupId } = await ec2Client.send(command);
    return GroupId;
};

const allocateIpAddress = async () => {
    const command = new AllocateAddressCommand({});
    const { PublicIp, AllocationId } = await ec2Client.send(command);
    return { PublicIp, AllocationId };
const command = new AuthorizeSecurityGroupIngressCommand({
  GroupId: securityGroupId,
  IpPermissions: [
    {
      IpProtocol: "tcp",
      FromPort: 22,
      ToPort: 22,
      IpRanges: [{ CidrIp: `${ipAddress}/32` }],
    },
  ],
});

await ec2Client.send(command);
return ipAddress;

const describeSecurityGroup = async (securityGroupName) => {
  const command = new DescribeSecurityGroupsCommand({
    GroupNames: [securityGroupName],
  });
  const { SecurityGroups } = await ec2Client.send(command);

  return SecurityGroups[0];
};

const getAmznLinux2AMIs = async () => {
  const AMIs = [];
  for await (const page of paginateGetParametersByPath({
    client: ssmClient,
  }, { Path: "/aws/service/ami-amazon-linux-latest" })) {
    page.Parameters.forEach((param) => {
      if (param.Name.includes("amzn2")) {
        AMIs.push(param.Value);
      }
    });
  }

  const imageDetails = [];
  for await (const page of paginateDescribeImages({
    client: ec2Client },
  { Path: "/aws/service/ami-amazon-linux-latest" })
  { page.Parameters.forEach((param) => {
      if (param.Name.includes("amzn2")) {
        AMIs.push(param.Value);
      }
    });
  })

  const imageDetails = [];
  for await (const page of paginateDescribeImages({
    client: ec2Client },
  { Path: "/aws/service/ami-amazon-linux-latest" })
  { page.Parameters.forEach((param) => {
      if (param.Name.includes("amzn2")) {
        AMIs.push(param.Value);
      }
    });
  })
{ ImageIds: AMIs },
)

imageDetails.push(...(page.Images || []));

const options = imageDetails.map(
  (image) => `${image.ImageId} - ${image.Description}`
);

/**
 * @type {number[]} 
*/
const [selectedIndex] = await promptToSelect(options);

return imageDetails[selectedIndex];

/**
 * @param {import('@aws-sdk/client-ec2').Image} imageDetails
 */
const getCompatibleInstanceTypes = async (imageDetails) => {
  const paginator = paginateDescribeInstanceTypes(
    { client: ec2Client, pageSize: 25 },
    {
      Filters: [
        {
          Name: "processor-info.supported-architecture",
          Values: [imageDetails.Architecture],
        },
        { Name: "instance-type", Values: ["*.micro", "*.small"] },
      ],
    },
  );

  const instanceTypes = [];

  for await (const page of paginator) {
    if (page.InstanceTypes.length) {
      instanceTypes.push(...(page.InstanceTypes || []));
    }
  }

  const instanceTypeList = instanceTypes.map(
    (type) => `${type.InstanceType} - Memory:${type.MemoryInfo.SizeInMiB}`
  ),
/**
 * @type {number[]} 
 */
const [selectedIndex] = await promptToSelect(
    instanceTypeList,
    "Select an instance type.",
);
return instanceTypes[selectedIndex];

const runInstance = async ({
    keyPairName,
    securityGroupId,
    imageId,
    instanceType,
}) => {
    const command = new RunInstancesCommand({
        KeyName: keyPairName,
        SecurityGroupIds: [securityGroupId],
        ImageId: imageId,
        InstanceType: instanceType,
        MinCount: 1,
        MaxCount: 1,
    });

    const { Instances } = await ec2Client.send(command);
    await waitUntilInstanceStatusOk(
        { client: ec2Client },
        { InstanceIds: [Instances[0].InstanceId] },
    );
    return Instances[0].InstanceId;
};

const describeInstance = async (instanceId) => {
    const command = new DescribeInstancesCommand({
        InstanceIds: [instanceId],
    });

    const { Reservations } = await ec2Client.send(command);
    return Reservations[0].Instances[0];
};
const displaySSHConnectionInfo = ({ publicIp, keyPairName }) => {
  return `ssh -i ${tmpDirectory}/${keyPairName}.pem ec2-user@$publicIp`;
};

canvas stopInstance = async (instanceId) => {
  const command = new StopInstancesCommand({ InstanceIds: [instanceId] });
  await ec2Client.send(command);
  await waitUntilInstanceStopped(
    { client: ec2Client },
    { InstanceIds: [instanceId] },
  );
};

const startInstance = async (instanceId) => {
  const startCommand = new StartInstancesCommand({ InstanceIds: [instanceId] });
  await ec2Client.send(startCommand);
  await waitUntilInstanceStatusOk(
    { client: ec2Client },
    { InstanceIds: [instanceId] },
  );
  return await describeInstance(instanceId);
};

const associateAddress = async ({ allocationId, instanceId }) => {
  const command = new AssociateAddressCommand({
    AllocationId: allocationId,
    InstanceId: instanceId,
  });

  const { AssociationId } = await ec2Client.send(command);
  return AssociationId;
};

const disassociateAddress = async (associationId) => {
  const command = new DisassociateAddressCommand({
    AssociationId: associationId,
  });
  await ec2Client.send(command);
};

const releaseAddress = async (allocationId) => {
  const command = new ReleaseAddressCommand({
    AllocationId: allocationId,
  });
try {
    await ec2Client.send(command);
    console.log(`# Address with allocation ID ${allocationId} released.
`);
    } catch (err) {
    console.log(err);
    }
}

const restartInstance = async (instanceId) => {
    console.log("Stopping instance.");
    await stopInstance(instanceId);
    console.log("Instance stopped.");
    console.log("Starting instance.");
    const { PublicIpAddress } = await startInstance(instanceId);
    return PublicIpAddress;
};

const terminateInstance = async (instanceId) => {
    const command = new TerminateInstancesCommand({
        InstanceIds: [instanceId],
    });

    try {
        await ec2Client.send(command);
        await waitUntilInstanceTerminated({
            client: ec2Client,
            InstanceIds: [instanceId],
        });
        console.log(`# Instance with ID ${instanceId} terminated.
`);
    } catch (err) {
        console.error(err);
    }
};

const deleteSecurityGroup = async (securityGroupId) => {
    const command = new DeleteSecurityGroupCommand({
        GroupId: securityGroupId,
    });

    try {
        await ec2Client.send(command);
        console.log(`# Security group ${securityGroupId} deleted.
`);
    } catch (err) {

```javascript
const deleteKeyPair = async (keyPairName) => {
    const command = new DeleteKeyPairCommand({
        KeyName: keyPairName,
    });

    try {
        await ec2Client.send(command);
        console.log(`# Key pair ${keyPairName} deleted.
`);
    } catch (err) {
        console.error(err);
    }
};

const deleteTemporaryDirectory = () => {
    try {
        rmSync(tmpDirectory, { recursive: true });
        console.log(`# Temporary directory ${tmpDirectory} deleted.
`);
    } catch (err) {
        console.error(err);
    }
};

export const main = async () => {
    const keyPairName = "ec2-scenario-key-pair";
    const securityGroupName = "ec2-scenario-security-group";

    let securityGroupId, ipAllocationId, publicIp, instanceId, associationId;

    console.log(wrapText("Welcome to the Amazon EC2 basic usage scenario.
"));

    try {
        // Prerequisites
        console.log("Before you launch an instance, you'll need a few things:",
            "\n - A Key Pair",
            "\n - A Security Group",
            "\n - An IP Address",
            "\n - An AMI",
            "\n - A compatible instance type",
```
await promptToContinue();

await createKeyPair(keyPairName);
securityGroupId = await createSecurityGroup(securityGroupName);
const { PublicIp, AllocationId } = await allocateIpAddress();
ipAllocationId = AllocationId;
publicIp = PublicIp;
const ipAddress = await authorizeSecurityGroupIngress(securityGroupId);

const { KeyName } = await describeKeyPair(keyPairName);
const { GroupName } = await describeSecurityGroup(securityGroupName);
console.log(`# created the key pair ${KeyName}.\n`);
console.log(`# created the security group ${GroupName}`, `and allowed SSH access from ${ipAddress} (your IP).\n`, `);
console.log(`# allocated ${publicIp} to be used for your EC2 instance.\n`);

await promptToContinue();

// Creating the instance
console.log(wrapText("Create the instance."));
console.log(`You get to choose which image you want. Select an amazon-linux-2 image from
the following:",
`);
const imageDetails = await getAmznLinux2AMIs();
const instanceTypeDetails = await getCompatibleInstanceTypes(imageDetails);
console.log("Creating your instance. This can take a few seconds.");
instanceId = await runInstance({
  keyPairName,
  securityGroupId,
  imageId: imageDetails.ImageId,
  instanceType: instanceTypeDetails.InstanceType,
});
const instanceDetails = await describeInstance(instanceId);
console.log(`# instance ${instanceId}.\n`);
console.log(instanceDetails);
console.log(`\nYou should now be able to SSH into your instance from another terminal:`,
`);
// Understanding the IP address.
console.log(wrapText("Understanding the IP address.");
console.log("When you stop and start an instance, the IP address will change. I'll restart your",
"instance for you. Notice how the IP address changes.");
const ipAddressAfterRestart = await restartInstance(instanceId);
console.log(`
Instance started. The IP address changed from ${instanceDetails.PublicIpAddress} to ${ipAddressAfterRestart}.

${displaySSHConnectionInfo({
  publicIp: ipAddressAfterRestart,
  keyPairName,
})}
`);
await promptToContinue();

console.log(`If you want to the IP address to be static, you can associate an allocated`,
`IP address to your instance. I allocated ${publicIp} for you earlier, and now I'll associate it to your instance.`);
associationId = await associateAddress({
  allocationId: ipAllocationId,
  instanceId,
});
console.log("Done. Now you should be able to SSH using the new IP.

${displaySSHConnectionInfo({ publicIp, keyPairName })}");
await promptToContinue();
console.log("I'll restart the server again so you can see the IP address remains the same.");
const ipAddressAfterAssociated = await restartInstance(instanceId);
console.log(`Done. Here's your SSH info. Notice the IP address hasn't changed.

```
${displaySSHConnectionInfo({
    publicIp: ipAddressAfterAssociated,
    keyPairName,
  })}
```
);
await promptToContinue();
} catch (err) {
    console.error(err);
} finally {
    // Clean up.
    console.log(wrapText("Clean up."));
    console.log("Now I'll clean up all of the stuff I created.");
    await promptToContinue();
    console.log("Cleaning up. Some of these steps can take a bit of time.");
    await disassociateAddress(associationId);
    await terminateInstance(instanceId);
    await releaseAddress(ipAllocationId);
    await deleteSecurityGroup(securityGroupId);
    deleteTemporaryDirectory();
    await deleteKeyPair(keyPairName);
    console.log("Done cleaning up. Thanks for staying until the end!",
              "If you have any feedback please use the feedback button in the docs",
              "or create an issue on GitHub.",
            );
}
};

- 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

Elastic Load Balancing 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.

*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 and cross-service examples.

*Scenarios* are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.

Each example includes a link to GitHub, 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.
Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```javascript
/*
 * Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
 * SPDX-License-Identifier: Apache-2.0
 */

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](#).
**Actions**

**Create a listener for a load balancer**

The following code example shows how to create a listener that forwards requests from an ELB load balancer to a target group.

```javascript
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](https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/ElasticLoadBalancingV2.html#CreateListener慣れ) in [AWS SDK for JavaScript API Reference](https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/ElasticLoadBalancingV2.html#CreateListener慣れ).

**Create a target group**

The following code example shows how to create an ELB target group.

**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws-samples/aws-code-examples).

Elastic Load Balancing

528
Create an Application Load Balancer

The following code example shows how to create an ELB Application Load Balancer.

```javascript
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.

**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.
```javascript
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](#).

**Delete a load balancer**

The following code example shows how to delete an ELB load balancer.

### SDK for JavaScript (v3)

```javascript
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](https://aws.amazon.com) in *AWS SDK for JavaScript API Reference*.

**Delete a target group**

The following code example shows how to delete an ELB target group.

**SDK for JavaScript (v3)**

```javascript
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](https://aws.amazon.com) 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](https://aws.amazon.com).
Describe target groups

The following code example shows how to describe specific target groups.

**SDK for JavaScript (v3)**

```javascript
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](#).
```

Get the endpoint of a load balancer

The following code example shows how to get the endpoint of an ELB load balancer.

**SDK for JavaScript (v3)**

```javascript
/*
 * Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
 * SPDX-License-Identifier: Apache-2.0
 */
```
AWS SDK for JavaScript

Developer Guide for SDK Version 3

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.
Get the health of a target group
The following code example shows how to get the health of instances in an ELB target group.
SDK for JavaScript (v3)
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(
Elastic Load Balancing

533


new DescribeTargetHealthCommand({
    TargetGroupArn: TargetGroups[0].TargetGroupArn,
    ik
});

- 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)**

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>There's more on GitHub. Find the complete example and learn how to set up and run in the <a href="https://github.com/aws/aws-sdk-js-v3">AWS Code Examples Repository</a>.</td>
</tr>
</tbody>
</table>

Run the interactive scenario at a command prompt.

```bash
#!/usr/bin/env node
```
import { Scenario, parseScenarioArgs, } from "@aws-sdk-examples/libs/scenario/index.js";

import { deploySteps } from "./steps-deploy.js";
import { demoSteps } from "./steps-demo.js";
import { destroySteps } from "./steps-destroy.js";

export const scenarios = {
  deploy: new Scenario("Resilient Workflow - Deploy", deploySteps, context),
  demo: new Scenario("Resilient Workflow - Demo", demoSteps, context),
  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.

/*@flow
* Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
* SPDX-License-Identifier: Apache-2.0
*/
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-sdk-examples/libs/scenario/index.js";
import { retry } from "@aws-sdk-examples/libs/utils/util-timers.js";

import { MESSAGES, NAMES, RESOURCES_PATH, ROOT } from "./constants.js";
import { initParamsSteps } from "./steps-reset-params.js";

/**
 * @type {import('@aws-sdk-examples/libs/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 },
    },

    writeFileSync(`${NAMES.keyPairName}.pem`, KeyMaterial, { mode: 0o600 });
});
```javascript
// Create instance policy
new CreatePolicyCommand({
    PolicyName: NAMES.instancePolicyName,
    PolicyDocument: readFileSync(
        join(RESOURCES_PATH, "instance_policy.json"),
    ),
}),

state.instancePolicyArn = Arn;

// Output created instance policy
new ScenarioOutput("createdInstancePolicy", (state) =>
    MESSAGES.createdInstancePolicy
        .replace("${INSTANCE_POLICY_NAME}", NAMES.instancePolicyName)
        .replace("${INSTANCE_POLICY_ARN}", state.instancePolicyArn),
),

// Create instance role
new ScenarioAction("createInstanceRole", () => {
    const client = new IAMClient({});
    return client.send(
        new CreateRoleCommand({
            RoleName: NAMES.instanceRoleName,
            AssumeRolePolicyDocument: readFileSync(
                join(ROOT, "assume-role-policy.json"),
            ),
        })),
    );
},

// Output created instance role
new ScenarioOutput("createdInstanceRole",
    MESSAGES.createdInstanceRole.replace(
        "${INSTANCE_ROLE_NAME}", NAMES.instanceRoleName,
    ),
),

// Attach policy to instance role
new ScenarioAction("attachingPolicyToRole", () => {
    return new ScenarioOutput(
        "attachingPolicyToRole",
        MESSAGES.attachingPolicyToRole.replace(
            "${INSTANCE_ROLE_NAME}", NAMES.instanceRoleName,
        ),
    );
}
```
new ScenarioAction("attachPolicyToRole", async (state) => {
  const client = new IAMClient({});
  await client.send(
    new AttachRolePolicyCommand({
      RoleName: NAMES.instanceRoleName,
      PolicyArn: state.instancePolicyArn,
    })),
  );
}),
new ScenarioOutput("attachedPolicyToRole",
  MESSAGES.attachedPolicyToRole
  .replace("${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 () => {
  // snippet-start:[javascript.v3.wkflw.resilient.CreateLaunchTemplate]
  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,
      },
    ),
  );
  // snippet-end:[javascript.v3.wkflw.resilient.CreateLaunchTemplate]
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",
/**
   * Created an auto-scaling group with the following details:
   *
   *  - Launch Template: ${launchTemplateName} (version: $Default)
   *  - Availability Zones: ${availabilityZoneNames}
   *  - Auto Scaling Group Name: ${autoScalingGroupName}
   *  - Min Size: 3
   *  - Max Size: 3
   *
   *  The auto-scaling is now active and ready to scale the EC2 instances.
   */
);
* @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) => {
  // snippet-start:[javascript.v3.wkflw.resilient.DescribeVpcs]
  const client = new EC2Client({});
  const { Vpcs } = await client.send(
    new DescribeVpcsCommand({
      Filters: [{ Name: "is-default", Values: ["true"] }],
    }),
  );
  // snippet-end:[javascript.v3.wkflw.resilient.DescribeVpcs]
  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) => {
  // snippet-start:[javascript.v3.wkflw.resilient.DescribeSubnets]
  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"] },
      ],
    }),
  );
  // snippet-end:[javascript.v3.wkflw.resilient.DescribeSubnets]
  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) => {
// snippet-start:[javascript.v3.wkflw.resilient.CreateTargetGroup]
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,
    ),
),
// snippet-end:[javascript.v3.wkflw.resilient.CreateTargetGroup]
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) => {
// snippet-start:[javascript.v3.wkflw.resilient.CreateLoadBalancer]
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] },
);
// snippet-end:[javascript.v3.wkflw.resilient.CreateLoadBalancer]
}),
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) => {
// snippet-start:[javascript.v3.wkflw.resilient.CreateListener]
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;

MESSAGES.createdLoadBalancerListener.replace(
  "$\{LB\_LISTENER\_ARN}\",
  state.loadBalancerListenerArn,
),

MESSAGES.attachingLoadBalancerTargetGroup
  .replace("$\{TARGET\_GROUP\_NAME}\", NAMES.loadBalancerTargetGroupName)
  .replace("$\{AUTO\_SCALING\_GROUP\_NAME}\", NAMES.autoScalingGroupName),

const client = new AutoScalingClient({});
await client.send(
  new AttachLoadBalancerTargetGroupsCommand({
    AutoScalingGroupName: NAMES.autoScalingGroupName,
    TargetGroupARNs: [state.targetGroupArn],
  }),
);

const client = new EC2Client({});
const { SecurityGroups } = await client.send(
  new DescribeSecurityGroupsCommand({
    SecurityGroupNames: [state.securityGroupName],
  }),
);
new DescribeSecurityGroupsCommand({
    Filters: [{ Name: "group-name", Values: ["default"] }],
}),
);
if (!SecurityGroups) {
    state.verifyInboundPortError = new Error(MESSAGES.noSecurityGroups);
} else {
    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,
    );
  }
}),
];

Create steps to run the demo.

/*
 * Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
 * SPDX-License-Identifier: Apache-2.0
 */
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,
"
import { ScenarioAction, ScenarioInput, ScenarioOutput, } from "@aws-sdk-examples/libs/scenario/scenario.js";
import { retry } from "@aws-sdk-examples/libs/utils/util-timers.js";

import { MESSAGES, NAMES, RESOURCES_PATH } from "./constants.js";
import { findLoadBalancer } from "./shared.js";

code
```javascript
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}`))
        } catch (error) {
            state.recommendation = "
        }
    } else {
        state.recommendation = "Failed to find load balancer.
    }
```
).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)}`
);  

const getHealthCheck = new ScenarioAction("getHealthCheck", async (state) => {
    // snippet-start:[javascript.v3.wkflw.resilient.DescribeTargetGroups]
    const client = new ElasticLoadBalancingV2Client({});
    const { TargetGroups } = await client.send(
        new DescribeTargetGroupsCommand({
            Names: [NAMES.loadBalancerTargetGroupName],
        })),
    );
    // snippet-end:[javascript.v3.wkflw.resilient.DescribeTargetGroups]

    // snippet-start:[javascript.v3.wkflw.resilient.DescribeTargetHealth]
    const { TargetHealthDescriptions } = await client.send(
        new DescribeTargetHealthCommand({
            TargetGroupArn: TargetGroups[0].TargetGroupArn,
        })),
    );
    // snippet-end:[javascript.v3.wkflw.resilient.DescribeTargetHealth]
    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:
    ${status}`;
},
{ preformatted: true },
);

const loadBalancerLoop = new ScenarioAction(  
  "loadBalancerLoop",
  getRecommendation.action,
  {
    whileConfig: {
      inputEquals: true,
      input: new ScenarioInput(  
        "loadBalancerCheck",
        MESSAGES.demoLoadBalancerCheck,
        {
          type: "confirm",
        },
      ),
      output: getRecommendationResult,
    },
  },
);

const healthCheckLoop = new ScenarioAction(  
  "healthCheckLoop",
  getHealthCheck.action,
  {
    whileConfig: {
      inputEquals: true,
      input: new ScenarioInput("healthCheck", MESSAGES.demoHealthCheck, {
        type: "confirm",
      },
    ),
    output: getHealthCheckResult,
  },
});

const statusSteps = [
  getRecommendation,
  getRecommendationResult,
  getHealthCheck,
getHealthCheckResult,
];

/**
 * @type {import('@aws-sdk-examples/libs/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];
// snippet-start:
[javascript.v3.wkflw.resilient.DescribeIamInstanceProfileAssociations]
const ec2Client = new EC2Client({});
const { IamInstanceProfileAssociations } = await ec2Client.send(
    new DescribeIamInstanceProfileAssociationsCommand({
        Filters: [
            { Name: "instance-id", Values: [state.targetInstance.InstanceId] },
        ],
    }));
// snippet-end:
[javascript.v3.wkflw.resilient.DescribeIamInstanceProfileAssociations]
state.instanceProfileAssociationId =
    IamInstanceProfileAssociations[0].AssociationId;
// snippet-start:
[javascript.v3.wkflw.resilient.ReplaceIamInstanceProfileAssociation]
await retry({ intervalInMs: 1000, maxRetries: 30 }, () =>
    ec2Client.send(
        new ReplaceIamInstanceProfileAssociationCommand({
            AssociationId: state.instanceProfileAssociationId,
            IamInstanceProfile: { Name: NAMES.ssmOnlyInstanceProfileName },
        }));
// snippet-end:
[javascript.v3.wkflw.resilient.ReplaceIamInstanceProfileAssociation]
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,
    }),
  );
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: Policy.Arn,
        }),
    );
}
RoleName: NAMES.ssmOnlyRoleName,
PolicyArn: "arn:aws:iam::aws:policy/AmazonSSMManagedInstanceCore",
);
// snippet-start:[javascript.v3.wkflw.resilient.CreateInstanceProfile]
const { InstanceProfile } = await iamClient.send(
    new CreateInstanceProfileCommand({
        InstanceProfileName: NAMES.ssmOnlyInstanceProfileName,
    })),
);
await waitUntilInstanceProfileExists(
    { client: iamClient },
    { InstanceProfileName: NAMES.ssmOnlyInstanceProfileName },
);
// snippet-end:[javascript.v3.wkflw.resilient.CreateInstanceProfile]
await iamClient.send(
    new AddRoleToInstanceProfileCommand({
        InstanceProfileName: NAMES.ssmOnlyInstanceProfileName,
        RoleName: NAMES.ssmOnlyRoleName,
    })),
);

return InstanceProfile;
}

Create steps to destroy all of the resources.

/*
 * Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
 * SPDX-License-Identifier: Apache-2.0
 */
import { unlinkSync } from "node:fs";

import { DynamoDBClient, DeleteTableCommand } from "@aws-sdk/client-dynamodb";
import {
    EC2Client,
    DeleteKeyPairCommand,
    DeleteLaunchTemplateCommand,
} from "@aws-sdk/client-ec2";
import {
    IAMClient,
    DeleteInstanceProfileCommand,
}
import { 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-sdk-examples/libs/scenario/index.js";
import { retry } from "@aws-sdk-examples/libs/utils/util-timers.js";

import { messages, names } from "./constants.js";
import { findLoadBalancer } from "./shared.js";

/**
 * @type {import('@aws-sdk-examples/libs/scenario.js').Step[]} 
 */
export const destroySteps = [
    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);
    const policy = await findPolicy(NAMES.instancePolicyName);
if (!policy) {
    state.detachPolicyFromRoleError = new Error(`Policy ${NAMES.instancePolicyName} not found.`);
} else {
    await client.send(
        new DetachRolePolicyCommand(
            {
                RoleName: NAMES.instanceRoleName,
                PolicyArn: policy.Arn,
            }
        ),
    );
    } catch (e) {
        state.detachPolicyFromRoleError = e;
    }
}),
new ScenarioOutput("detachedPolicyFromRole", (state) => {
    if (state.detachPolicyFromRoleError) {
        console.error(state.detachPolicyFromRoleError);
        return MESSAGES.detachedPolicyFromRole
            .replace("${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({

Elastic Load Balancing
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 {
   // snippet-start:[javascript.v3.wkflw.resilient.DeleteInstanceProfile]
   const client = new IAMClient({});
   await client.send(
     new DeleteInstanceProfileCommand({
       InstanceProfileName: NAMES.instanceProfileName,
     }),
   );
   // snippet-end:[javascript.v3.wkflw.resilient.DeleteInstanceProfile]
 } 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("deleteLaunchTemplate", async (state) => {
    const client = new EC2Client({});
    try {
        // snippet-start:[javascript.v3.wkflw.resilient.DeleteLaunchTemplate]
        await client.send(
            new DeleteLaunchTemplateCommand({
                LaunchTemplateName: NAMES.launchTemplateName,
            }),
            );
        // snippet-end:[javascript.v3.wkflw.resilient.DeleteLaunchTemplate]
        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("deleteAutoScalingGroup", async (state) => {
    try {
        await terminateGroupInstances(NAMES.autoScalingGroupName);
        await retry({ intervalInMs: 30000, 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("deleteLoadBalancer", async (state) => {
    try {
        // snippet-start:[javascript.v3.wkflw.resilient.DeleteLoadBalancer]
        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.");
            }
        });
        // snippet-end:[javascript.v3.wkflw.resilient.DeleteLoadBalancer]
    } 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,

});

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;
}

// snippet-end:[javascript.v3.wkflw.resilient.DeleteTargetGroup]

new ScenarioAction("detachSsmOnlyRoleFromProfile", async (state) => {
    try {
        const client = new IAMClient({});
        await client.send(
            new DeleteRoleFromProfileCommand({
                Name: NAMES.loadBalancerProfileName,
                RoleArn: NAMES.loadBalancerRoleArn,
            }),
        );
    } catch (e) {
        state.detachSsmOnlyRoleFromProfileError = e;
    }
}

// snippet-end:[javascript.v3.wkflw.resilient.DeleteProfile]
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("${PROFILE_NAME}", "AmazonSSMManagedInstanceProfile");
  } else {
    return MESSAGES.detachedSsmOnlyInstanceProfile
      .replace("${PROFILE_NAME}", "AmazonSSMManagedInstanceProfile");
  }
})}
"${INSTANCE_PROFILE_NAME}",
    NAMES.ssmOnlyInstanceProfileName,
); } else {
    return MESSAGES.deletedSsmOnlyInstanceProfile.replace(
        "${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;
    }
}),
)
```javascript
/**
 * @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 (e) {
        state.deleteSsmOnlyRoleError = e;
    }
}
```
```
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;
        }
    }
}

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;
        }
    }
}
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 and cross-service examples.

*Scenarios* are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.

Each example includes a link to GitHub, where you can find instructions on how to set up and run the code in context.

**Topics**

- *Actions*

## Actions

### Add a target

The following code example shows how to add a target to an Amazon EventBridge event.

**SDK for JavaScript (v3)**

```javascript
import {
  EventBridgeClient,
  PutTargetsCommand,
} from "@aws-sdk/client-eventbridge";
```

---

*Note*

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws/aws-sdk-js-v3).
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.
// 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.

Create a rule

The following code example shows how to create an Amazon EventBridge rule.
Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws-samples/aws-code-examples).

Import the SDK and client modules and call the API.

```javascript
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:
```
• For API details, see **PutRule** in *AWS SDK for JavaScript API Reference*.

**SDK for JavaScript (v2)**

```javascript
// 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*.
**Send events**

The following code example shows how to send Amazon EventBridge events.

**SDK for JavaScript (v3)**

```javascript
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',
      ],
    },
  ],
};
• For API details, see PutEvents in AWS SDK for JavaScript API Reference.

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.

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 and cross-service examples.

Scenarios are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.

Each example includes a link to GitHub, 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.
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('
');
    console.log('Job names: ');
    console.log(formattedJobNames);
    return JobNames;
};

- For API details, see ListJobs in AWS SDK for JavaScript API Reference.

Topics
- Actions
- Scenarios

Actions

Create a crawler

The following code example shows how to create an AWS Glue crawler.
const createCrawler = (name, role, dbName, tablePrefix, s3TargetPath) => {
  const client = new GlueClient({ region: DEFAULT_REGION });

  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.

Create a job definition

The following code example shows how to create an AWS Glue job definition.
const createJob = (name, role, scriptBucketName, scriptKey) => {
  const client = new GlueClient({ region: DEFAULT_REGION });

  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](AWS SDK for JavaScript API Reference).

**Delete a crawler**

The following code example shows how to delete an AWS Glue crawler.

**SDK for JavaScript (v3)**

```javascript
const deleteCrawler = (crawlerName) => {
  const client = new GlueClient({ region: DEFAULT_REGION });

  const command = new DeleteCrawlerCommand({
    Name: crawlerName,
  });

  return client.send(command);
};
```

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://aws.amazon.com/code-examples/).
Delete a database from the Data Catalog

The following code example shows how to delete a database from the AWS Glue Data Catalog.

SDK for JavaScript (v3)

```javascript
const deleteDatabase = (databaseName) => {
    const client = new GlueClient({ region: DEFAULT_REGION });

    const command = new DeleteDatabaseCommand({
        Name: databaseName,
    });

    return client.send(command);
};
```

For API details, see `DeleteDatabase` in AWS SDK for JavaScript API Reference.

Delete a job definition

The following code example shows how to delete an AWS Glue job definition and all associated runs.

```javascript
const deleteDatabase = (databaseName) => {
    const client = new GlueClient({ region: DEFAULT_REGION });

    const command = new DeleteDatabaseCommand({
        Name: databaseName,
    });

    return client.send(command);
};
```

For API details, see `DeleteDatabase` in AWS SDK for JavaScript API Reference.
const deleteJob = (jobName) => {
    const client = new GlueClient({ region: DEFAULT_REGION });

    const command = new DeleteJobCommand({
        JobName: jobName,
    });

    return client.send(command);
};

• For API details, see DeleteJob in AWS SDK for JavaScript API Reference.

Delete a table from a database

The following code example shows how to delete a table from an AWS Glue Data Catalog database.

const deleteTable = (databaseName, tableName) => {
    const client = new GlueClient({ region: DEFAULT_REGION });

    const command = new DeleteTableCommand({
        DatabaseName: databaseName,
    });

    return client.send(command);
};

• For API details, see DeleteTable in AWS SDK for JavaScript API Reference.
• For API details, see [DeleteTable](https://docs.aws.amazon.com/sdk-for-javascript/v3/api/latest/services/glue.html#Glue.Client.DeleteTable) in *AWS SDK for JavaScript API Reference*.

**Get a crawler**

The following code example shows how to get an AWS Glue crawler.

**SDK for JavaScript (v3)**

```javascript
const getCrawler = (name) => {
    const client = new GlueClient({ region: DEFAULT_REGION });

    const command = new GetCrawlerCommand({
        Name: name,
    });

    return client.send(command);
};
```

• For API details, see [GetCrawler](https://docs.aws.amazon.com/sdk-for-javascript/v3/api/latest/services/glue.html#Glue.Client.GetCrawler) in *AWS SDK for JavaScript API Reference*.

**Get a database from the Data Catalog**

The following code example shows how to get a database from the AWS Glue Data Catalog.
Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```javascript
const getDatabase = (name) => {
    const client = new GlueClient({ region: DEFAULT_REGION });

    const command = new GetDatabaseCommand({
        Name: name,
    });

    return client.send(command);
};
```

- For API details, see GetDatabase in AWS SDK for JavaScript API Reference.

Get a job run

The following code example shows how to get an AWS Glue job run.

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```javascript
const getJobRun = (jobName, jobRunId) => {
    const client = new GlueClient({ region: DEFAULT_REGION });
    const command = new GetJobRunCommand({
        JobName: jobName,
        RunId: jobRunId,
    });

    return client.send(command);
};
```
const getDatabases = () => {
    const client = new GlueClient({ region: DEFAULT_REGION });

    const command = new GetDatabasesCommand({});

    return client.send(command);
};

• For API details, see GetDatabases in AWS SDK for JavaScript API Reference.

Get databases from the Data Catalog

The following code example shows how to get a list of databases from the AWS Glue Data Catalog.

SDK for JavaScript (v3)

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 job from the Data Catalog

The following code example shows how to get a job from the AWS Glue Data Catalog.
SDK for JavaScript (v3)

Note
There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```javascript
const getJob = (jobName) => {
    const client = new GlueClient({ region: DEFAULT_REGION });

    const command = new GetJobCommand({
        JobName: jobName,
    });

    return client.send(command);
};
```

- For API details, see GetJob in AWS SDK for JavaScript API Reference.

Get runs of a job

The following code example shows how to get runs of an AWS Glue job.

SDK for JavaScript (v3)

Note
There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```javascript
const getJobRuns = (jobName) => {
    const client = new GlueClient({ region: DEFAULT_REGION });

    const command = new GetJobRunsCommand({
        JobName: jobName,
    });

    return client.send(command);
};
```
const getTables = (databaseName) => {
  const client = new GlueClient({ region: DEFAULT_REGION });

  const command = new GetTablesCommand({
    DatabaseName: databaseName,
  });

  return client.send(command);
};

- For API details, see GetTables in AWS SDK for JavaScript API Reference.

**Get tables from a database**

The following code example shows how to get tables from a database in the AWS Glue Data Catalog.

**SDK for JavaScript (v3)**

- For API details, see GetJobRuns in AWS SDK for JavaScript API Reference.

**List job definitions**

The following code example shows how to list AWS Glue job definitions.
SDK for JavaScript (v3)

**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws-samples/aws-code-examples).

```
const listJobs = () => {
    const client = new GlueClient({ region: DEFAULT_REGION });
    const command = new ListJobsCommand({});
    return client.send(command);
};
```

- For API details, see [ListJobs](https://aws-sdk.github.io/aws-sdk-js/2.0/api/latest/index.html#GlueClient.ListJobs) in *AWS SDK for JavaScript API Reference*.

**Start a crawler**

The following code example shows how to start an AWS Glue crawler.

SDK for JavaScript (v3)

**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws-samples/aws-code-examples).

```
const startCrawler = (name) => {
    const client = new GlueClient({ region: DEFAULT_REGION });
    const command = new StartCrawlerCommand({
        Name: name,
    });
    return client.send(command);
};
```
Start a job run

The following code example shows how to start an AWS Glue job run.

SDK for JavaScript (v3)

```javascript
const startJobRun = (jobName, dbName, tableName, bucketName) => {
    const client = new GlueClient({ region: DEFAULT_REGION });

    const command = new StartJobRunCommand({
        JobName: jobName,
        Arguments: {
            "--input_database": dbName,
            "--input_table": tableName,
            "--output_bucket_url": `s3://${bucketName}/`,
        },
    });

    return client.send(command);
};
```

Scenarios

Get started with crawlers and jobs

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)

```javascript
const createCrawler = (name, role, dbName, tablePrefix, s3TargetPath) => {
  const client = new GlueClient({ region: DEFAULT_REGION });

  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({ region: DEFAULT_REGION });

  const command = new GetCrawlerCommand({
    Name: name,
  });

  return client.send(command);
};
```

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.
Name: name,
});

return client.send(command);
};

const startCrawler = (name) => {
    const client = new GlueClient({ region: DEFAULT_REGION });

    const command = new StartCrawlerCommand({
        Name: name,
    });

    return client.send(command);
};

const crawlerExists = async ({ getCrawler }, crawlerName) => {
    try {
        await getCrawler(crawlerName);
        return true;
    } catch {
        return false;
    }
};

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 };}

/**
const getDatabase = (name) => {
  const client = new GlueClient({ region: DEFAULT_REGION });

  const command = new GetDatabaseCommand({
    Name: name,
  });

  try {
    const { Database } = await client.send(command);
  } catch (error) {
    throw new Error(`Database with name ${name} not found.`);
  }

  return { ...Database, Name: name,};
};

List information about databases and tables in your AWS Glue Data Catalog.

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 };};
const getTables = (databaseName) => {
    const client = new GlueClient({ region: DEFAULT_REGION });

    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 }
    };

const makeGetTablesStep =
    ({ getTables }) =>
    async (context) => {
        const { TableList } = await getTables(process.env.DATABASE_NAME);
        log(`Tables:`);
        log(TableList.map((table) => `  • ${table.Name}
`));
        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({ region: DEFAULT_REGION });

    const command = new CreateJobCommand({

const startJobRun = (jobName, dbName, tableName, bucketName) => {
    const client = new GlueClient({ region: DEFAULT_REGION });
    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 };
```javascript
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);
};

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(
        region=${DEFAULT_REGION}&tab=objects 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.

callback getJobRuns = (jobName) => {
    const client = new GlueClient({ region: DEFAULT_REGION });
    const command = new GetJobRunsCommand({
        JobName: jobName,
    });

    return client.send(command);
};

callback getJobRun = (jobName, jobRunId) => {
    return 

```
const client = new GlueClient({ region: DEFAULT_REGION });
const command = new GetJobRunCommand({
    JobName: jobName,
    RunId: jobRunId,
});

return client.send(command);
};

const logJobRunDetails = async (getJobRun, jobName, jobRunId) => {
    const { JobRun } = await getJobRun(jobName, jobRunId);
    log(JobRun, { type: "object" });
};

const makePickJobRunStep =
    ({ getJobRuns, getJobRun }) =>
    async (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({ region: DEFAULT_REGION });

    const command = new DeleteJobCommand({
        JobName: jobName,
    });
```
const deleteTable = (databaseName, tableName) => {
    const client = new GlueClient({ region: DEFAULT_REGION });

    const command = new DeleteTableCommand({
        DatabaseName: databaseName,
        Name: tableName,
    });

    return client.send(command);
};

const deleteDatabase = (databaseName) => {
    const client = new GlueClient({ region: DEFAULT_REGION });

    const command = new DeleteDatabaseCommand({
        Name: databaseName,
    });

    return client.send(command);
};

const deleteCrawler = (crawlerName) => {
    const client = new GlueClient({ region: DEFAULT_REGION });

    const command = new DeleteCrawlerCommand({
        Name: crawlerName,
    });

    return client.send(command);
};

const handleDeleteJobs = async (deleteJobFn, jobNames, context) => {
    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.");
    }
const makeCleanUpJobsStep =
({ listJobs, deleteJob }) =>
async (context) => {
  const { JobNames } = await listJobs();
  if (JobNames.length > 0) {
    await handleDeleteJobs(deleteJob, JobNames, context);
  }

  return { ...context };
};

const deleteTables = (deleteTable, databaseName, tableNames) =>
Promise.all(
  tableNames.map((tableName) =>
    deleteTable(databaseName, tableName).catch(console.error)
  )
);
} else {
    log("Deleting tables.");
    await deleteTables(deleteTable, process.env.DATABASE_NAME, tableNames);
    log("Tables deleted.", { type: "success" });
}

return { ...context };}

const deleteDatabases = (deleteDatabase, databaseNames) =>
    Promise.all(
        databaseNames.map((dbName) => deleteDatabase(dbName).catch(console.error))
    );

const makeCleanUpDatabasesStep =
    ({ getDatabases, deleteDatabase }) =>
    async (context) => {
        const { DatabaseList } = await getDatabases();

        if (DatabaseList.length > 0) {
            const { dbNames } = await context.prompter.prompt({
                name: "dbNames",
                type: "checkbox",
                message: "Let's clean up databases. Select databases to delete."
            });
            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 {
        
    } catch (error) {
        log(error.message);
    }

    return { ...context };}

);
```javascript
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](#)
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 and cross-service examples.

*Scenarios* are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.

Each example includes a link to GitHub, where you can find instructions on how to set up and run the code in context.

**Get started**

**Hello HealthImaging**

The following code example shows how to get started using HealthImaging.

**SDK for JavaScript (v3)**

```javascript
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
    //   },
```
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

#### Add a tag to a resource

The following code example shows how to add a tag to a HealthImaging resource.

**SDK for JavaScript (v3)**

```javascript
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"
 *   }
 * /

  tags = {})
```
```javascript
) => {
    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](#).

### Copy an image set

The following code example shows how to copy a HealthImaging image set.

#### SDK for JavaScript (v3)

Utility function to copy an image set.

```javascript
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.
*/

export const copyImageSet = async (
  datastoreId = "xxxxxxxxxxxxxx",
  imageSetId = "xxxxxxxxxxxxxx",
  sourceVersionId = "1",
  destinationImageSetId = "",
  destinationVersionId = ""
) => {
  const params = {
    datastoreId: datastoreId,
    sourceImageSetId: imageSetId,
    copyImageSetInformation: {
      sourceImageSet: { latestVersionId: sourceVersionId },
    },
  };
  if (destinationImageSetId !== "" && destinationVersionId !== "") {
    params.copyImageSetInformation.destinationImageSet = {
      imageSetId: destinationImageSetId,
      latestVersionId: destinationVersionId,
    };
  }

  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: 'xxxxxxxxxxxxxx',
  //     destinationImageSetProperties: {
  //       createdAt: 2023-09-27T19:46:21.824Z,
  //     }
  // },
//     imageSetId: 'xxxxxxxxxxxxxxxx',
//     imageSetState: 'LOCKED',
//     imageSetWorkflowStatus: 'COPYING',
//     latestVersionId: '1',
//   },
//   sourceImageSetProperties: {
//     createdAt: 2023-09-22T14:49:26.427Z,
//     imageSetId: 'xxxxxxxxxxxxxxxx',
//     imageSetState: 'LOCKED',
//     imageSetWorkflowStatus: 'COPYING_WITH_READ_ONLY_ACCESS',
//     latestVersionId: '4',
//   }
// }
// }
// return response;
});
For API details, see [CopyImageSet](https://aws.amazon.com) 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](https://aws.amazon.com).

---

**Create a data store**

The following code example shows how to create a HealthImaging data store.

**SDK for JavaScript (v3)**

```javascript
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: 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',
  //   datastoreStatus: 'CREATING'
  // }
```

---
Delete a data store

The following code example shows how to delete a HealthImaging data store.

**SDK for JavaScript (v3)**

```javascript
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: 'f5be409-678d-48c9-9173-9a001e1eb1',
    //     extendedRequestId: undefined,
    //     cfId: undefined,
    //     attempts: 1,
    //     totalRetryDelay: 0
    //   },
    //   datastoreId: 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',
    //   datastoreStatus: 'DELETING'
    // }
};
```

- 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](#).
Delete an image set

The following code example shows how to delete a HealthImaging image set.

**SDK for JavaScript (v3)**

```javascript
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 = 'xxxxxxxxxxxxxxxx', imageSetId = 'xxxxxxxxxxxxxxxx') => {
    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,
    //   },
    // }

    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](#).
• For API details, see [DeleteImageSet](https://aws-sdk.github.io/aws-sdk-for-javascript/v3/api/index.html#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](https://github.com/aws/aws-sdk-js-v3).  

Get an image frame

The following code example shows how to get an image frame.

**SDK for JavaScript (v3)**

```javascript
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"

  return response;
);
```javascript
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](#).

### Get data store properties

The following code example shows how to get HealthImaging data store properties.

**SDK for JavaScript (v3)**
```javascript
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,
    //         datastoreId: 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',
    //         datastoreName: 'my_datastore',
    //         datastoreStatus: 'ACTIVE',
    //         updatedAt: 2023-08-04T18:50:36.239Z
    //     }
    // }
    return response['datastoreProperties'];
};
```

- For API details, see [GetDatastore](https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/service-docs/medical-imaging-client.html#GetDatastoreCommand) in [AWS SDK for JavaScript API Reference](https).

**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https).
Get image set properties

The following code example shows how to get HealthImaging image set properties.

**SDK for JavaScript (v3)**

```javascript
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 = "xxxxxxxxxxxxxxx",
    imageSetId = "xxxxxxxxxxxxxxx",
    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: 'xxxxxxxxxxxxxxx',
 //xxxxxxxxxxxxxxxxxxxxx/imageset/xxxxxxxxxxxxxxxxxxxxxx',
    //     imageSetId: 'xxxxxxxxxxxxxxx',
    //     imageSetState: 'ACTIVE',
```
Get import job properties

The following code example shows how to get the import job properties.

**SDK for JavaScript (v3)**

```javascript
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 = "xxxxxxxxxxxxxxxxxxxx", jobId = "xxxxxxxxxxxxxxxxxxxx") => {
    const response = await medicalImagingClient.send(new GetDICOMImportJobCommand({ datastoreId: datastoreId, jobId: jobId }));
    console.log(response);
    // {"$metadata": {
    //     httpStatusCode: 200,
```
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.

Get metadata for an image set

The following code example shows how to get metadata for a HealthImaging image set.

SDK for JavaScript (v3)

Utility function to get image set metadata.

```javascript
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 = "xxxxxxxxxxxxxx",
    imagesetId = "xxxxxxxxxxxxxx",
    versionID = ""
) => {
    const params = { 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: '521b274-30ff-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.

```javascript
try {
    await getImageSetMetadata(
        "metadata.json.gzip",
        "12345678901234567890123456789012",
        "12345678901234567890123456789012"
    );
} catch (err) {
    console.log("Error", err);
}
```

Get image set metadata with version.

```javascript
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](#).

---

### Import bulk data into a data store

The following code example shows how to import bulk data into a HealthImaging data store.
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: 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',
  //   jobid: 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',
  //   jobStatus: 'SUBMITTED',
  // },
List data stores

The following code example shows how to list HealthImaging data stores.

**SDK for JavaScript (v3)**

```javascript
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);
  }
```

- For API details, see [StartDICOMImportJob](https://aws.amazon.com/documentation/sdk-for-javascript/api-reference/) 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](https://github.com/aws-samples/aws-code-examples).
// {// 
//   '$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,
//xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',
//       datastoreId: 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',
//       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.

---

**List image set versions**

The following code example shows how to list HealthImaging image set versions.

**SDK for JavaScript (v3)**

```javascript
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 = "xxxxxxxxxxxx", imageSetId = "xxxxxxxxxxxx") => {
    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: 'xxxxxxxxxxxxxxxxxxxxx',
    //             imageSetState: 'ACTIVE',
    //             versionId: '1'
    //     ]
    // }
// }
List import jobs for a data store

The following code example shows how to list import jobs for a HealthImaging data store.

**SDK for JavaScript (v3)**

```javascript
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 = "xxxxxxxxxxxxxxxxxx") => {
  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"]);
  }

  return jobSummaries;
};
```

• 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](#).
The following code example shows how to list tags for a HealthImaging resource.

**SDK for JavaScript (v3)**

```javascript
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:datatstore/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.

Remove a tag from a resource

The following code example shows how to remove a tag from a HealthImaging resource.

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.
 */
  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](https://sdk.amazonaws.com) in [AWS SDK for JavaScript API Reference](https).

**Note**
There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https).
Search image sets

The following code example shows how to search HealthImaging image sets.

SDK for JavaScript (v3)

The utility function for searching image sets.

```javascript
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.
 */
export const searchImageSets = async (datastoreId = 'xxxxxxxx', filters = []) => {
  const paginatorConfig = {
    client: medicalImagingClient,
    pageSize: 50,
  };

  const commandParams = {
    datastoreId: datastoreId,
    searchCriteria: {
      filters,
    },
  };

  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,
```
Use case #1: EQUAL operator.

```javascript
const datastoreId = "12345678901234567890123456789012";

try {
  const filters = [
    {
      values: [{ DICOMPatientId: "9227465" }],
      operator: "EQUAL",
    },
  ];

  await searchImageSets(datastoreId, filters);
} catch (err) {
  console.error(err);
}
```

Use case #2: BETWEEN operator using DICOMStudyDate and DICOMStudyTime.

```javascript
const datastoreId = "12345678901234567890123456789012";

try {
  
  
} catch (err) {
  
  
}
```
```javascript
const filters = [
    {
        values: [
            {
                DICOMStudyDateAndTime: {
                    DICOMStudyDate: "19900101",
                    DICOMStudyTime: "000000",
                },
            },
            {
                DICOMStudyDateAndTime: {
                    DICOMStudyDate: "20230901",
                    DICOMStudyTime: "000000",
                },
            },
        ],
        operator: "BETWEEN",
    },
];

await searchImageSets(datastoreId, filters);

try {
    const filters = [
        {
            values: [
                { createdAt: new Date("1985-04-12T23:20:50.52Z") },
                { createdAt: new Date("2023-09-12T23:20:50.52Z") },
            ],
            operator: "BETWEEN",
        },
    ];

    await searchImageSets(datastoreId, filters);
} catch (err) {
    console.error(err);
}
```

Use case #3: BETWEEN operator using createdAt. Time studies were previously persisted.
Update image set metadata

The following code example shows how to update HealthImaging image set metadata.

SDK for JavaScript (v3)

```javascript
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.
 */
export const updateImageSetMetadata = async (datastoreId = "xxxxxxxxxx",
  imageSetId = "xxxxxxxxxx",
  latestVersionId = "1",
  updateMetadata = '{}') => {

  const response = await medicalImagingClient.send(
    new UpdateImageSetMetadataCommand({
      datastoreId: datastoreId,
      imageSetId: imageSetId,
      latestVersionId: latestVersionId,
      updateImageSetMetadataUpdates: updateMetadata
    })
  );
  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: 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',
//     imageSetId: 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',
//     imageSetState: 'LOCKED',
//     imageSetWorkflowStatus: 'UPDATING',
//     latestVersionId: '4',
//     updatedAt: 2023-09-27T19:41:43.494Z
// }
// return response;

Encode the metadata.

const updatableAttributes =
JSON.stringify({
    "SchemaVersion": 1.1,
    "Patient": {
        "DICOM": {
            "PatientName": "Garcia^Gloria"
        }
    }
});

const updateMetadata = {
    "DICOMUpdates": {
        "updatableAttributes":
            new TextEncoder().encode(updatableAttributes)
    }
};

await updateImageSetMetadata("12345678901234567890123456789012", "12345678901234567890123456789012", "1", updateMetadata);
Scenarios

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"}
 */
```
    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/123456789012345678901234567890123456789012";
  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 (
  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 <i>AWS SDK for JavaScript API Reference</i>.
  - ListTagsForResource
  - TagResource
  - UntagResource

<i>Note</i>
There's more on GitHub. Find the complete example and learn how to set up and run in the <a>AWS Code Examples Repository</a>.
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.

```javascript
try {
  const imagesetArn = 
    "arn:aws:medical-imaging:us-east-1:12345678901234567890123456789012/ 
    imageset/12345678901234567890123456789012";
  const tags = {
    Deployment: "Development",
  };
  await tagResource(imagesetArn, tags);
} catch (e) {
  console.log(e);
}
```

The utility function for tagging a resource.

```javascript
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 (
  xxx",
  tags = {}
) => {
  const response = await medicalImagingClient.send(
    new TagResourceCommand({ resourceArn: resourceArn, tags: tags })
  );
  console.log(response);
  // {
```
To list tags for an image set.

```javascript
try {
    const imagesetArn = 
        "arn:aws:medical-imaging:us-east-1:123456789012:123456789012/12345678901234567890123456789012/ 
            imageset/12345678901234567890123456789012";
    const { tags } = await listTagsForResource(imagesetArn);
    console.log(tags);
} catch (e) {
    console.log(e);
}
```

The utility function for listing a resource's tags.

```javascript
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:datatstore/def/imageset/ghi"
) => {
    const response = await medicalImagingClient.send(
        new ListTagsForResourceCommand({ resourceArn: resourceArn })
    );
    return response;
};
```
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:datstore/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 (healthImaging
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 and cross-service examples.

Scenarios are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.

Each example includes a link to GitHub, 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)

```
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](https://aws.amazon.com/sdk-for-javascript/api-reference/API_IAM_ListPolicies/) in *AWS SDK for JavaScript API Reference*.

### Topics

- **Actions**
- **Scenarios**

### Actions

**Attach a policy to a role**

The following code example shows how to attach an IAM policy to a role.

**SDK for JavaScript (v3)**

```javascript
// Import necessary modules
import { AttachRolePolicyCommand, IAMClient } from '@aws-sdk/client-iam';

// Create an IAM client instance
const client = new IAMClient({});

// Attach the policy
/**
 * 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.

```javascript
import { AttachRolePolicyCommand, IAMClient } from '@aws-sdk/client-iam';

const client = new IAMClient({});

/**
 * Attach a policy to a role
 */
```
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)

ℹ️ 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 = {
    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.

Attach an inline policy to a role

The following code example shows how to attach an inline policy to an IAM role.

SDK for JavaScript (v3)

```javascript
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;
The following code example shows how to create an AWS Identity and Access Management (IAM) SAML provider.

SDK for JavaScript (v3)

```javascript
import { CreateSAMLProviderCommand, IAMClient } from '@aws-sdk/client-iam';
import { readFileSync } from 'fs';
import * as path from 'path';
import { dirnameFromMetaUrl } from '@aws-sdk-examples/libs/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
 */
```
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](aws-sdk): `CreateSAMLProvider` in AWS SDK for JavaScript API Reference.

Create a group

The following code example shows how to create an IAM group.

**SDK for JavaScript (v3)**

```javascript
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);
};
```

[Note]

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws/aws-sdk-code-examples).
Create a policy

The following code example shows how to create an IAM policy.

**SDK for JavaScript (v3)**

```javascript
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 response;
};
```

For API details, see [CreateGroup](#) 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](#).
return client.send(command);
;

- For more information, see [AWS SDK for JavaScript Developer Guide](https://aws.amazon.com/sdk-for-javascript/).
- For API details, see [CreatePolicy](https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/IAM.html#CreatePolicy-property) in [AWS SDK for JavaScript API Reference](https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/IAM.html).

**SDK for JavaScript (v2)**

ℹ️ **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws/aws-sdk-js-examples).

```javascript
// 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.

Create a role

The following code example shows how to create an IAM role.

SDK for JavaScript (v3)

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](https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/index.html/CreateRole) in [AWS SDK for JavaScript API Reference](https).

### Create a service-linked role

The following code example shows how to create an IAM service-linked role.

**SDK for JavaScript (v3)**

```javascript
import { CreateServiceLinkedRoleCommand, IAMClient } from '@aws-sdk/client-iam';

const client = new IAMClient({});

const client = new IAMClient({});

const serviceLinkedRole = new CreateServiceLinkedRoleCommand({
  ServiceName: 'Lambda',
  RoleName: 'LambdaServiceRole',
});

const response = await client.send(serviceLinkedRole);
```

*Note*

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https).
createServiceLinkedRole = async (serviceName) => {
  const command = new CreateServiceLinkedRoleCommand(
    { AWSServiceName: serviceName },
  );

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

For API details, see CreateServiceLinkedRole in AWS SDK for JavaScript API Reference.

Create a user

The following code example shows how to create an IAM user.

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.

SDK for JavaScript (v3)

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.

```javascript
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](https://docs.aws.amazon.com/sdk-for-javascript/v3/developerguide/)
- For API details, see [CreateUser](https://docs.aws.amazon.com/sdk-for-javascript/v3/api/client-iam.htm#iam-createuser-command) in [AWS SDK for JavaScript API Reference](https://docs.aws.amazon.com/sdk-for-javascript/v3/api/

**SDK for JavaScript (v2)**

```javascript
// 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') {
```
Create an access key

The following code example shows how to create an IAM access key.

```javascript
iam.createUser(params, function(err, data) {
    if (err) {
        console.log("Error", err);
    } else {
        console.log("Success", data);
    }
});

if (err) {
    console.log("Error", err);
} 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](#).

**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](#).

**SDK for JavaScript (v3)**

**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.

```javascript
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)

ℹ️ 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.
Create an alias for an account

The following code example shows how to create an alias for an IAM account.

SDK for JavaScript (v3)

```javascript
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](https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/).
- For API details, see [CreateAccountAlias](https://docs.aws.amazon.com/sdk-for-javascript/v3/api/latest/index.html) in [AWS SDK for JavaScript API Reference](https://docs.aws.amazon.com/sdk-for-javascript/v3/api/latest/).

SDK for JavaScript (v2)

```javascript
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
// 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.

Create an instance profile

The following code example shows how to create an IAM instance profile.

SDK for JavaScript (v3)

ℹ️ 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 },
);
• For API details, see [CreateInstanceProfile](https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/iam.html#_createinstanceprofile___) in *AWS SDK for JavaScript API Reference*.

## Delete SAML provider

The following code example shows how to delete an AWS Identity and Access Management (IAM) SAML provider.

### SDK for JavaScript (v3)

```javascript
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](https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/iam.html#deletesamlprovider___) 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](https://github.com/aws/aws-sdk-js-client-v3/blob/master/examples/iam/delete-saml-provider.ts).
Delete a group

The following code example shows how to delete an IAM group.

SDK for JavaScript (v3)

```javascript
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](https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AutoGeneratedFiles/AWS.IAM.Client.html#DeleteGroupCommand) in *AWS SDK for JavaScript API Reference*.

Delete a policy

The following code example shows how to delete an IAM policy.
Delete the policy.

```javascript
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](#).

Delete a role

The following code example shows how to delete an IAM role.

SDK for JavaScript (v3)
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](aws-sdk-api-docs) in *AWS SDK for JavaScript API Reference*.

### Delete a role policy

The following code example shows how to delete an IAM role policy.

#### SDK for JavaScript (v3)

```javascript
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.

Delete a server certificate

The following code example shows how to delete an IAM server certificate.

SDK for JavaScript (v3)

```javascript
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.

```javascript
// 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.

Delete a service-linked role

The following code example shows how to delete an IAM service-linked role.

SDK for JavaScript (v3)

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*.

**Delete a user**

The following code example shows how to delete an IAM user.

⚠️ **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](#).

**SDK for JavaScript (v3)**

ℹ️ **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.

```javascript
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) {

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

- For more information, see [AWS SDK for JavaScript Developer Guide](https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/).
- For API details, see [DeleteUser](https://docs.aws.amazon.com/sdk-for-javascript/v3/api/latest/lib/iam.html) in [AWS SDK for JavaScript API Reference](https://docs.aws.amazon.com/sdk-for-javascript/v3/api/latest/).

### Delete an access key

The following code example shows how to delete an IAM access key.

⚠️ **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](https://aws.amazon.com/identity-center/).

### SDK for JavaScript (v3)

ℹ️ **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws-code-examples).

Delete the access key.

```javascript
import { DeleteAccessKeyCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 * Delete an access key.
 */
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](https://docs.aws.amazon.com/js SDK for JavaScript Developer Guide).
• For API details, see [DeleteAccessKey](https://docs.aws.amazon.com/js SDK for JavaScript API Reference).

**SDK for JavaScript (v2)**

ℹ️ **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws/aws-sdk-js-v3).

```javascript
// 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);
    }
});
```
For more information, see [AWS SDK for JavaScript Developer Guide](https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/).


### Delete an account alias

The following code example shows how to delete an IAM account alias.

#### SDK for JavaScript (v3)

```javascript
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](https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/).

**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws/aws-sdk-js/tree/master/code_examples/iam).

```javascript
// 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](https://sdk.amazonaws.com/).
- For API details, see [DeleteAccountAlias](https://sdk.amazonaws.com/API/2010-05-08/index.html#DeleteAccountAlias) in [AWS SDK for JavaScript API Reference](https://sdk.amazonaws.com/).

### Delete an instance profile

The following code example shows how to delete an IAM instance profile.

**SDK for JavaScript (v3)**

**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws/aws-sdk-js/tree/master/code_examples/iam).
const client = new IAMClient({});
await client.send(
    new DeleteInstanceProfileCommand({
        InstanceProfileName: NAMES.instanceProfileName,
    })),
);

• For API details, see DeleteInstanceProfile in AWS SDK for JavaScript API Reference.

**Detach a policy from a role**

The following code example shows how to detach an IAM policy from a role.

**SDK for JavaScript (v3)**

```javascript
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);
```

**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 information, see [AWS SDK for JavaScript Developer Guide](https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/).

For API details, see [DetachRolePolicy](https://docs.aws.amazon.com/sdk-for-javascript/v3/api/AWS.IAM.html#IAM-DetachRolePolicy) in [AWS SDK for JavaScript API Reference](https://docs.aws.amazon.com/sdk-for-javascript/v3/api/AWS.IAM.html).

### SDK for JavaScript (v2)

```javascript
// 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 = {
                    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");
                    }
                });
            }
        });
    }
});
```
Get a policy

The following code example shows how to get an IAM policy.

SDK for JavaScript (v3)

```javascript
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 DetachRolePolicy in AWS SDK for JavaScript API Reference.
Get a role

The following code example shows how to get an IAM role.

```javascript
// 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 = {
};

iam.getPolicy(params, function(err, data) {
    if (err) {
        console.log("Error", err);
    } else {
        console.log("Success", data.Policy.Description);
    }
});
```
Get the role.

```javascript
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.

Get a server certificate

The following code example shows how to get an IAM server certificate.

SDK for JavaScript (v3)
Get a server certificate.

```javascript
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](https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/).

**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](https://github.com/aws/aws-sdk-js-v3).
For more information, see [AWS SDK for JavaScript Developer Guide](https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/).

Get a service-linked role's deletion status

The following code example shows how to get an AWS Identity and Access Management (IAM) service-linked role's deletion status.

**SDK for JavaScript (v3)**

```
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,
  });
  client.send(command, async (err, data) => {
    if (err) {
      console.log("Error", err);
    } else {
      console.log("Success", data);
    }
  });
```
For API details, see [GetServiceLinkedRoleDeletionStatus](#) in [AWS SDK for JavaScript API Reference](#).

### Get data about the last use of an access key

The following code example shows how to get data about the last use of an IAM access key.

⚠️ **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](#).

### SDK for JavaScript (v3)

**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.

```javascript
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,
    })
    return client.send(command);
};
```
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.

```javascript
// 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);
    }
});
```
Get the account password policy

The following code example shows how to get the IAM account password policy.

SDK for JavaScript (v3)

```javascript
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.

List SAML providers

The following code example shows how to list SAML providers for IAM.

- For more information, see AWS SDK for JavaScript Developer Guide.
- For API details, see GetAccessKeyLastUsed in AWS SDK for JavaScript API Reference.
List the SAML providers.

```javascript
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](#).

## List a user's access keys

The following code example shows how to list a user's IAM access keys.

⚠️ **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](#).
List the access keys.

```javascript
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,
        }),
      );
    }
  }
}
```
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.

```javascript
// 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.

List account aliases

The following code example shows how to list IAM account aliases.

SDK for JavaScript (v3)

```javascript
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,
                }));
        }
    }
}
```

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 information, see AWS SDK for JavaScript Developer Guide.

• For API details, see ListAccountAliases in AWS SDK for JavaScript API Reference.

SDK for JavaScript (v2)

```javascript
// 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.
List groups

The following code example shows how to list IAM groups.

SDK for JavaScript (v3)

```javascript
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,
                }));
        }
    }
}
```

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 inline policies for a role

The following code example shows how to list inline policies for an IAM role.

SDK for JavaScript (v3)

```javascript
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,
  });
}
```

For API details, see ListGroups in AWS SDK for JavaScript API Reference.
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.

List policies

The following code example shows how to list IAM policies.

SDK for JavaScript (v3)

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 functions to simplify this.

```javascript
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].

**List policies attached to a role**

The following code example shows how to list policies attached to an IAM role.
List the policies that are attached to a role.

```javascript
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,
                }));
        }
    }
```
• For API details, see ListAttachedRolePolicies in AWS SDK for JavaScript API Reference.

List roles

The following code example shows how to list IAM roles.

**SDK for JavaScript (v3)**

```javascript
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.
 * 
 * @type {import('@aws-sdk/client-iam').ListRolesCommandOutput | undefined}
 */
export async function* listRoles() {
    const command = new ListRolesCommand({
        MaxItems: 10,
    });

    /**
     * @type {import("@aws-sdk/client-iam").ListRolesCommandOutput | undefined}
     */
```

---

**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.
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.

List server certificates

The following code example shows how to list IAM server certificates.

SDK for JavaScript (v3)

### 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.

```javascript
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 functions to simplify this.

```javascript
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](https://aws.amazon.com/sdk-for-javascript/).

For API details, see [ListServerCertificates](https://docs.aws.amazon.com/AWSJavaScriptSDK/v3/latest/index.html#list-server-certificates) in [AWS SDK for JavaScript API Reference](https://aws.amazon.com/api/).

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](https://github.com/aws-samples/aws-sdk-code-examples).

```javascript
// 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](https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/).
- For API details, see [ListServerCertificates](https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/IAM.html#ListServerCertificates) in [AWS SDK for JavaScript API Reference](https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/IAM.html).

### List users

The following code example shows how to list IAM users.

⚠️ **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](https://aws.amazon.com/iam/).

### SDK for JavaScript (v3)

ℹ️ **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws-samples/aws-sdk-js-developer-guide).

List the users.

```javascript
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);
    });
  }
});
Update a server certificate

The following code example shows how to update an IAM server certificate.

SDK for JavaScript (v3)

```javascript
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](https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/).

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```javascript
// 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.

Update a user

The following code example shows how to update an IAM user.
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.

SDK for JavaScript (v3)

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.

```javascript
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](https://github.com/aws-samples/aws-code-examples).

```javascript
// 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](https).
- For API details, see [UpdateUser](https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/Auto-generated/UpdateUser.html) in [AWS SDK for JavaScript API Reference](https).

**Update an access key**

The following code example shows how to update an IAM access key.
AWS SDK for JavaScript
Developer Guide for SDK Version 3

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.

SDK for JavaScript (v3)

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.

```javascript
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.

```javascript
// 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.
Upload a server certificate

The following code example shows how to upload an AWS Identity and Access Management (IAM) server certificate.

SDK for JavaScript (v3)

```javascript
import { UploadServerCertificateCommand, IAMClient } from '@aws-sdk/client-iam';
import { readFileSync } from 'fs';
import { dirnameFromMetaUrl } from '@aws-sdk-examples/libs/utils/util-fs.js';
import * as path from 'path';

const client = new IAMClient({});

/**
 * The certificate body and private key were generated with the following command.
 * ```
 * 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 certBody = readFileSync(
    path.join(
        dirnameFromMetaUrl(import.meta.url),
        "../../../../resources/sample_files/sample_cert.pem",
    ),
);

const privateKey = readFileSync(
    path.join(
        dirnameFromMetaUrl(import.meta.url),
        "../../../../resources/sample_files/sample_private_key.pem",
    ),
);
```

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.
export const uploadServerCertificate = (certificateName) => {
  const command = new UploadServerCertificateCommand({
    ServerCertificateName: certificateName,
    CertificateBody: certBody.toString(),
    PrivateKey: privateKey.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.
Run the interactive scenario at a command prompt.

```bash
#!/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-sdk-examples/libs/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
```
create steps to deploy all of the resources.

```javascript
/*
 * Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
 * SPDX-License-Identifier: Apache-2.0
 */
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-sdk-examples/libs/scenario/index.js"
import { retry } from "@aws-sdk-examples/libs/utils/util-timers.js"

import { MESSAGES, NAMES, RESOURCES_PATH, ROOT } from "./constants.js"
import { initParamsSteps } from "./steps-reset-params.js"

/**
 * @type {import('@aws-sdk-examples/libs/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 waitUntilInstanceProfile Exists( 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 () => {
  // snippet-start:[javascript.v3.wkflw.resilient.CreateLaunchTemplate]
  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: {
          InstanceType: "t3.micro",
          ImageId: Parameter.Value,
          IamInstanceProfile: { Name: NAMES.instanceProfileName },
          UserData: readFileSync(
            join(RESOURCES_PATH, "server_startup_script.sh"),
            ).toString("base64"),
          KeyName: NAMES.keyPairName,
        },
      }));
});
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("${AVAILABILITY_ZONE_NAMES}",
  state.availabilityZoneNames.join(" ",
),
),
new ScenarioInput("confirmContinue", MESSAGES.confirmContinue, {
  type: "confirm",
}),
new ScenarioOutput("loadBalancer", MESSAGES.loadBalancer),
new ScenarioOutput("gettingVpc", MESSAGES.gettingVpc),
new ScenarioAction("getVpc", async (state) => {
  // snippet-start:[javascript.v3.wkflw.resilient.DescribeVpcs]
  const client = new EC2Client({});
  const { Vpcs } = await client.send(
    new DescribeVpcsCommand({
      Filters: [{ Name: "is-default", Values: ["true"] }],
    }),
  );
  // snippet-end:[javascript.v3.wkflw.resilient.DescribeVpcs]
  state.defaultVpc = Vpcs[0].VpcId;
}),
new ScenarioOutput("gotVpc", (state) =>
  MESSAGES.gotVpc.replace("${VPC_ID}", state.defaultVpc),
),
new ScenarioOutput("gettingSubnets", MESSAGES.gettingSubnets),
new ScenarioAction("getSubnets", async (state) => {
  // snippet-start:[javascript.v3.wkflw.resilient.DescribeSubnets]
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"] },
        ],
    )
);
// snippet-end:[javascript.v3.wkflw.resilient.DescribeSubnets]
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) => {
    // snippet-start:[javascript.v3.wkflw.resilient.CreateTargetGroup]
    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,
        )
    );
});
// snippet-end:[javascript.v3.wkflw.resilient.CreateTargetGroup]
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) => {
  // snippet-start:[javascript.v3.wkflw.resilient.CreateLoadBalancer]
  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] },
  );
  // snippet-end:[javascript.v3.wkflw.resilient.CreateLoadBalancer]
});
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) => {
  // snippet-start:[javascript.v3.wkflw.resilient.CreateListener]
  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 },
      ],
    }));
  // snippet-end:[javascript.v3.wkflw.resilient.CreateListener]
  const listener = Listeners[0];
  state.loadBalancerListenerArn = listener.ListenerArn;
});
// snippet-end:[javascript.v3.wkflw.resilient.CreateListener]
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) => {
  // snippet-start:[javascript.v3.wkflw.resilient.AttachTargetGroup]
  const client = new AutoScalingClient({});
  await client.send(
    new AttachLoadBalancerTargetGroupsCommand({
      AutoScalingGroupName: NAMES.autoScalingGroupName,
      TargetGroupARNs: [state.targetGroupArn],
    })),
  );
  // snippet-end:[javascript.v3.wkflw.resilient.AttachTargetGroup]
}),

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];

  /**
   * @param {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",
      }),
    );
  }
);
Create steps to run the demo.

/*
* Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
* SPDX-License-Identifier: Apache-2.0
*/
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-sdk-examples/libs/scenario/scenario.js";
import { retry } from "@aws-sdk-examples/libs/utils/util-timers.js";
```javascript
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:
${JSON.stringify(state.recommendation, null, 2)}`,
    { preformatted: true },
);

const getHealthCheck = new ScenarioAction("getHealthCheck", async (state) => {
    // snippet-start:[javascript.v3.wkflw.resilient.DescribeTargetGroups]
    const client = new ElasticLoadBalancingV2Client({});
    const { TargetGroups } = await client.send(new DescribeTargetGroupsCommand({
        Names: [NAMES.loadBalancerTargetGroupName],
    })),
    // snippet-end:[javascript.v3.wkflw.resilient.DescribeTargetGroups]

    // snippet-start:[javascript.v3.wkflw.resilient.DescribeTargetHealth]
    const { TargetHealthDescriptions } = await client.send(new DescribeTargetHealthCommand({
        TargetGroupArn: TargetGroups[0].TargetGroupArn,
    })),
    // snippet-end:[javascript.v3.wkflw.resilient.DescribeTargetHealth]
});
```

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:
        ${status}
    `;
},
{ preformatted: true },
);

const loadBalancerLoop = new ScenarioAction("loadBalancerLoop",
    getRecommendation.action,
    {
        whileConfig: {
            inputEquals: true,
            input: new ScenarioInput("loadBalancerCheck",
                MESSAGES.demoLoadBalancerCheck,
                {
                    type: "confirm",
                },
                output: getRecommendationResult,
            },
        },
    });

const healthCheckLoop = new ScenarioAction("healthCheckLoop",
    getHealthCheck.action,
    {
        whileConfig: {
            inputEquals: true,
        },
    });
input: new ScenarioInput("healthCheck", MESSAGES.demoHealthCheck, {
  type: "confirm",
}),
output: getHealthCheckResult,
},
],
);

const statusSteps = [
getRecommendation,
getRecommendationResult,
getHealthCheck,
getHealthCheckResult,
];

/**
 * @type {import('@aws-sdk-examples/libs/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) =>

IAM
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];
  // snippet-start:
  [javascript.v3.wkflw.resilient.DescribeIamInstanceProfileAssociations]
  const ec2Client = new EC2Client({});
  const { IamInstanceProfileAssociations } = await ec2Client.send(
    new DescribeIamInstanceProfileAssociationsCommand({
      Filters: [
        { Name: "instance-id", Values: [state.targetInstance.InstanceId] },
      ],
    }),
  );
  // snippet-end:
  [javascript.v3.wkflw.resilient.DescribeIamInstanceProfileAssociations]
  state.instanceProfileAssociationId =
    IamInstanceProfileAssociations[0].AssociationId;
  // snippet-start:
  [javascript.v3.wkflw.resilient.ReplaceIamInstanceProfileAssociation]
  await retry({ intervalInMs: 1000, maxRetries: 30 }, () =>
    ec2Client.send(
      new ReplaceIamInstanceProfileAssociationCommand({
        AssociationId: state.instanceProfileAssociationId,
        IamInstanceProfile: { Name: NAMES.ssmOnlyInstanceProfileName },
      }),
    ),
  );
// snippet-end:
[javascript.v3.wkflw.resilient.ReplaceIamInstanceProfileAssociation]

    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,
    IAM
{ 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",
          },
        ],
      }),
    }));
}
Create steps to destroy all of the resources.

/*
* Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
* SPDX-License-Identifier: Apache-2.0
*/
import { unlinkSync } from "node:fs";

import { DynamoDBClient, DeleteTableCommand } from "@aws-sdk/client-dynamodb";
import {
    EC2Client,
    DeleteKeyPairCommand,
    DeleteLaunchTemplateCommand,
} from "@aws-sdk/client-ec2";
import {
    IAMClient,
    DeleteInstanceProfileCommand,
    RemoveRoleFromInstanceProfileCommand,
    DeletePolicyCommand,
    DeleteRoleCommand,
    DetachRolePolicyCommand,
    paginateListPolicies,
} from "@aws-sdk/client-iam";
import {
    AutoScalingClient,
    DeleteAutoScalingGroupCommand,
    TerminateInstanceInAutoScalingGroupCommand,
    UpdateAuto ScalingGroupCommand,
    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-sdk-examples/libs/scenario/index.js";
import { retry } from "@aws-sdk-examples/libs/utils/util-timers.js";

import { MESSAGES, NAMES } from "./constants.js";
import { findLoadBalancer } from "./shared.js";

/**
 * @type {import('@aws-sdk-examples/libs/scenario.js').Step[]}
export const destroySteps = [
    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 ${NAMES.instancePolicyName} not found.`
            );
        } else {
            await client.send(
                new DetachRolePolicyCommand({
                    RoleName: NAMES.instanceRoleName,
                    PolicyArn: policy.Arn,
                }),
            );
        }
    } catch (e) {
        state.detachPolicyFromRoleError = e;
    }
});
new ScenarioOutput("detachedPolicyFromRole", (state) => {
    if (state.detachPolicyFromRoleError) {
        console.error(state.detachPolicyFromRoleError);
        return MESSAGES.detachedPolicyFromRoleError
            .replace("${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("${INSTANCE_PROFILE_NAME}", NAMES.instanceProfileName)
```javascript
.replace("${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 {
        // snippet-start:[javascript.v3.wkflw.resilient.DeleteInstanceProfile]
        const client = new IAMClient({});
        await client.send(
            new DeleteInstanceProfileCommand({
                InstanceProfileName: NAMES.instanceProfileName,
            })),
            );
        // snippet-end:[javascript.v3.wkflw.resilient.DeleteInstanceProfile]
    } catch (e) {
```
```javascript
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("deleteLaunchTemplate", async (state) => {
  const client = new EC2Client({});
  try {
    // snippet-start:[javascript.v3.wkflw.resilient.DeleteLaunchTemplate]
    await client.send(
      new DeleteLaunchTemplateCommand({
        LaunchTemplateName: NAMES.launchTemplateName,
      }),
    );
    // snippet-end:[javascript.v3.wkflw.resilient.DeleteLaunchTemplate]
  } 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("deleteAutoScalingGroup", async (state) => {
    try {
        await terminateGroupInstances(NAMES.autoScalingGroupName);
        await retry({ intervalInMs: 30000, 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("deleteLoadBalancer", async (state) => {
    try {
        // snippet-start:[javascript.v3.wkflw.resilient.DeleteLoadBalancer]
        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.");
            }
        });
        // snippet-end:[javascript.v3.wkflw.resilient.DeleteLoadBalancer]
    } 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) => {
    // snippet-start:[javascript.v3.wkflw.resilient.DeleteTargetGroup]
    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;
    }
    // snippet-end:[javascript.v3.wkflw.resilient.DeleteTargetGroup]
}),
new ScenarioOutput("deleteLoadBalancerTargetGroupResult", (state) => {
    if (state.deleteLoadBalancerTargetGroupError) {
        console.error(state.deleteLoadBalancerTargetGroupError);
        return MESSAGES.deleteLoadBalancerTargetGroupError.replace(  
            "${TARGET_GROUP_NAME}",
            NAMES.loadBalancerTargetGroupName,
        );
    } else {
        return MESSAGES.deletedLoadBalancer.replace(  
            "${LB_NAME}",
            NAMES.loadBalancerName,
        );
    }
});
); } 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) {

IAM
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,
    );
    }
});
"${POLICY_NAME}",
  NAMES.ssmOnlyPolicyName,
);}
},
new ScenarioAction("deleteSsmOnlyRole", async (state) => {
  try {
    const iamClient = new IAMClient({});
    await iamClient.send(
      new DeleteRoleCommand({
        RoleName: NAMES.ssmOnlyRoleName,
      }),
    );
  } catch (e) {
    state.deleteSsmOnlyRoleError = e;
  }
}),
new ScenarioOutput("deleteSsmOnlyRoleResult", (state) => {
  if (state.deleteSsmOnlyRoleError) {
    console.error(state.deleteSsmOnlyRoleError);
    return MESSAGES.deleteSsmOnlyRoleError.replace("${ROLE_NAME}",
      NAMES.ssmOnlyRoleName,
    );
  } else {
    return MESSAGES.deletedSsmOnlyRole.replace("${ROLE_NAME}",
      NAMES.ssmOnlyRoleName,
    );
  }
}),
];

/**
 * @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;
    }
}
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
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](https://aws.amazon.com/iam/).

- 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)

ℹ️ Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws/aws-sdk-js-v3).
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.

```javascript
import {
    CreateUserCommand,
    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-sdk-examples/libs/utils/util-timers.js';

// Set the parameters.
const iamClient = new IAMClient({});
const userName = "test_name";
const policyName = "test_policy";
const roleName = "test_role";

export const main = async () => {
    // Create a user. The user has no permissions by default.
    const { User } = await iamClient.send(
        new CreateUserCommand({ UserName: userName }),
    );

    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 }),
    );

    // Assume the role.
    const assumeRoleResponse = await iamClient.send(
        new AssumeRoleCommand({ RoleName: roleName }),
    );

    // Get temporary credentials using the assumed role.
    const assumeRoleResult = assumeRoleResponse.$response;
    const assumedRoleCredentials = assumeRoleResult.Credentials;

    // Use the temporary credentials to list buckets.
    const s3Client = new S3Client({
        region: assumedRoleCredentials.Region,
        credentials: assumedRoleCredentials,
    });

    const listBucketsResponse = await s3Client.send(
        new ListBucketsCommand()
    );

    console.log("Listed buckets:");
    listBucketsResponse.Buckets.forEach(bucket => console.log(bucket.Name));

    // Clean up.
    await iamClient.send(new DeleteAccessKeyCommand({ AccessKeyId: createAccessKeyResponse.AccessKeyId }));
    await iamClient.send(new DeleteUserCommand({ UserName: userName }));
    await iamClient.send(new DeleteRoleCommand({ RoleName: roleName }));
}
```
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.Arnn,  
    RoleName: Role.RoleName,
  })),
);

await iamClient.send(
  new DeletePolicyCommand({
    PolicyArn: listBucketPolicy.Arnn,
  })),
);

await iamClient.send(
  new DeleteRoleCommand({
    RoleName: Role.RoleName,
  })),
);

await iamClient.send(
  new DeleteAccessKeyCommand({
    UserName: userName,
    AccessKeyId,
  })),
);

await iamClient.send(
  new DeleteUserCommand({
    UserName: userName,
  })),
);
```javascript
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

**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 and cross-service examples.

*Scenarios* are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.
Each example includes a link to GitHub, 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)

```javascript
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

Actions

Create a function

The following code example shows how to create a Lambda function.

SDK for JavaScript (v3)

```javascript
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);
};
```

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 API details, see CreateFunction in AWS SDK for JavaScript API Reference.

Delete a function

The following code example shows how to delete a Lambda function.
SDK for JavaScript (v3)

Note
There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```javascript
/**
 * @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](https://aws.amazon.com/sdk-for-javascript/api-reference/) in *AWS SDK for JavaScript API Reference*.

Get a function

The following code example shows how to get a Lambda function.

SDK for JavaScript (v3)

Note
There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```javascript
const getFunction = (funcName) => {
    const client = new LambdaClient({});
    const command = new GetFunctionCommand({ FunctionName: funcName });
    return client.send(command);
};
```
Invoke a function

The following code example shows how to invoke a Lambda function.

**SDK for JavaScript (v3)**

```javascript
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*.

List functions

The following code example shows how to list Lambda functions.

• For API details, see **GetFunction** 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*. 

---

Copyright © 2023, Amazon Web Services, Inc. or its affiliates. All rights reserved.
SDK for JavaScript (v3)

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.

Update function code

The following code example shows how to update Lambda function 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.

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
For API details, see UpdateFunctionConfiguration in AWS SDK for JavaScript API Reference.

### SDK for JavaScript (v3)

```javascript
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

#### 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.

```javascript
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.

```javascript
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.

```javascript
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.

```javascript
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({});

/**
 */
```javascript
* @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](#)

### 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 and cross-service examples.

*Scenarios* are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.
Each example includes a link to GitHub, where you can find instructions on how to set up and run the code in context.

Topics

• Actions

Actions

Create a batch interface job

The following code example shows how to create a Amazon Personalize batch interface job.

SDK for JavaScript (v3)

```javascript
// 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 */
umResults: 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.

Create a batch segment job

The following code example shows how to create a Amazon Personalize batch segment job.

SDK for JavaScript (v3)

⚠️ 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](#).

### Create a campaign

The following code example shows how to create a Amazon Personalize campaign.
SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```javascript
// 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.
Create a dataset

The following code example shows how to create a Amazon Personalize dataset.

SDK for JavaScript (v3)

```javascript
// 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](https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/api/index.html>CreateDataset) in [AWS SDK for JavaScript API Reference](https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/api/index.html).
Create a dataset export job

The following code example shows how to create a Amazon Personalize dataset export job.

SDK for JavaScript (v3)

```javascript
// 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);  
  }
```

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 dataset group

The following code example shows how to create a Amazon Personalize dataset group.

SDK for JavaScript (v3)

```javascript
// 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);
    }
};
```

• For API details, see [CreateDatasetExportJob](#) 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](#).
Create a domain dataset group.

```javascript
// 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](#).

**Create a dataset import job**

The following code example shows how to create a Amazon Personalize dataset import job.
// 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.
Create a domain schema

The following code example shows how to create a Amazon Personalize domain schema.

SDK for JavaScript (v3)

```javascript
// 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.

Create a filter

The following code example shows how to create a Amazon Personalize filter.

SDK for JavaScript (v3)

// 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 {
        // There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.
const response = await personalizeClient.send(new CreateFilterCommand(createFilterParam));
console.log("Success", response);
return response; // For unit tests.
} catch (err) {
  console.log("Error", err);
}
// For API details, see CreateFilter in AWS SDK for JavaScript API Reference.

Create a recommender

The following code example shows how to create a Amazon Personalize recommender.

SDK for JavaScript (v3)

// 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 () => {

// There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

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*.

Create a schema

The following code example shows how to create a Amazon Personalize schema.

SDK for JavaScript (v3)

⚠️ Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws-code-examples/).</ref>
```javascript
try {
    const response = await personalizeClient.send(new CreateSchemaCommand(createSchemaParam));
    console.log("Success", response);
    return response; // For unit tests.
} catch (err) {
    console.log("Error", err);
}
```

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();

- For API details, see CreateSchema in AWS SDK for JavaScript API Reference.

Create a solution

The following code example shows how to create a Amazon Personalize solution.

SDK for JavaScript (v3)

⚠️ 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.

Create a solution version

The following code example shows how to create a Amazon Personalize solution.

SDK for JavaScript (v3)

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*.

Create an event tracker

The following code example shows how to create a Amazon Personalize event tracker.

**SDK for JavaScript (v3)**

```javascript
// 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"});
```

- 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 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](#).

### 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 and cross-service examples.

**Scenarios** are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.

Each example includes a link to GitHub, where you can find instructions on how to set up and run the code in context.

**Topics**

• **Actions**
Actions

Import items into a dataset

The following code example shows how to incrementally import items into an Amazon Personalize Events dataset.

SDK for JavaScript (v3)

```
// 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);
    }
}
```
For API details, see [PutItems](https://aws.amazon.com/sdk-for-js/api-reference/) in *AWS SDK for JavaScript API Reference*.

**Import real-time interaction event data**

The following code example shows how to import real-time interaction event data into Amazon Personalize Events.

**SDK for JavaScript (v3)**

```javascript
// 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 */,
    },
  ]
};
```

---

**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws-samples/aws-code-examples).
```
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](https://aws-sdk.github.io/aws-sdk-js-v3/API.html#PutEvents) in *AWS SDK for JavaScript API Reference*.

**Incrementally import a user**

The following code example shows how to incrementally import a user into Amazon Personalize Events.

**SDK for JavaScript (v3)**

```
// 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"});
```

**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws-samples/amazon-personalize-code-examples).
// 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 and cross-service examples.

Scenarios are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.

Each example includes a link to GitHub, where you can find instructions on how to set up and run the code in context.
Topics

- Actions

Actions

Get recommendations (custom dataset group)

The following code example shows how to get Amazon Personalize Runtime ranked recommendations.

SDK for JavaScript (v3)

```
// 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);
  }
}
```
Get recommendations from a recommender (domain dataset group)

The following code example shows how to get Amazon Personalize Runtime Runtime recommendations.

SDK for JavaScript (v3)

```javascript
// 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));
  }
};
```

For API details, see `GetPersonalizedRanking` in *AWS SDK for JavaScript API Reference*.
Get recommendation with a filter (custom dataset group).

```javascript
import { GetRecommendationsCommand } from '@aws-sdk/client-personalize-runtime';
import { personalizeRuntimeClient } from './libs/personalizeClients.js';

// 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.

```javascript
import { GetRecommendationsCommand } from '@aws-sdk/client-personalize-runtime';
import { personalizeRuntimeClient } from './libs/personalizeClients.js';

// 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();
```
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 and cross-service examples.
Scenarios are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.

Each example includes a link to GitHub, where you can find instructions on how to set up and run the code in context.

Topics

- Actions

Actions

Send email and text messages

The following code example shows how to send email and text messages with Amazon Pinpoint.

SDK for JavaScript (v3)

```javascript
import { PinpointClient } from '@aws-sdk/client-pinpoint';
// Set the AWS Region.
const REGION = "us-east-1";
// Set the MediaConvert Service Object
const pinClient = new PinpointClient({ region: REGION });
export { pinClient };
```

Create the client in a separate module and export it.

```javascript
// 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 = `<html>
<head></head>
<body>
<h1>Amazon Pinpoint Test (SDK for JavaScript in Node.js)</h1>
<p>This email was sent with the Amazon Pinpoint Email API using the AWS SDK for JavaScript in Node.js.</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,
                    },
                },
            },
        },
    },
};
Send an SMS message.

```javascript
const run = async () => {
  try {
    const data = await pinClient.send(new SendMessagesCommand(params));

    const { MessageResponse: { Result }, } = data;

    const recipientResult = Result[toAddress];

    if (recipientResult.StatusCode !== 200) {
      throw new Error(recipientResult.StatusMessage);
    } else {
      console.log(recipientResult.MessageId);
    }
  } catch (err) {
    console.log(err.message);
  }
};
run();
```

// Import required AWS SDK clients and commands for Node.js
import { SendMessagesCommand } from "@aws-sdk/client-pinpoint";
import { pinClient } from "./libs/pinClient.js";

("use strict");

/* 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., XXXXXXXX66e4e9986478cXXXXXXXXX

/* 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]: {
            }}
    }}
const run = async () => {
  try {
    const data = await pinClient.send(new SendMessagesCommand(params));
    return data; // For unit tests.
    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 = `<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 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.sendMessages(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.sendMessages(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 SendMessages in AWS SDK for JavaScript API Reference.

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 and cross-service examples.

Scenarios are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.

Each example includes a link to GitHub, where you can find instructions on how to set up and run the code in context.

Topics
- Actions
Actions

Create a cluster

The following code example shows how to create an Amazon Redshift cluster.

SDK for JavaScript (v3)

```javascript
// 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.
};

Create the cluster.
```

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.
```
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*.

### Delete a cluster

The following code example shows how to delete an Amazon Redshift cluster.

#### SDK for JavaScript (v3)

```
const { RedshiftClient } = require("@aws-sdk/client-redshift");
// Set the AWS Region.
const REGION = "REGION";
// Set the Redshift Service Object
const redshiftClient = new RedshiftClient({ region: REGION });
export { redshiftClient };
```

---

For API details, see [CreateCluster](#) in *AWS SDK for JavaScript API Reference*.

Delete a cluster

The following code example shows how to delete an Amazon Redshift cluster.

SDK for JavaScript (v3)

- For API details, see [CreateCluster](#) in *AWS SDK for JavaScript API Reference*.

Create the client.
Create the cluster.

```javascript
// 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](https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/redshift-client-deletecluster.html) in *AWS SDK for JavaScript API Reference*.

Describe your clusters

The following code example shows how to describe your Amazon Redshift clusters.

**SDK for JavaScript (v3)**

**:info: Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws-samples/aws-sdk-code-examples).

Create the client.

```javascript
const { RedshiftClient } = require('@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.

Modify a cluster

The following code example shows how to modify an Amazon Redshift cluster.

SDK for JavaScript (v3)

⚠️ 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.

```javascript
const { RedshiftClient } = require('@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.

```javascript
// 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](https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/redshift-apis.html#redshift-commands-modify-cluster) in *AWS SDK for JavaScript API Reference*.

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.
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 and cross-service examples.

Scenarios are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.

Each example includes a link to GitHub, 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)

```javascript
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('
'));
   return Buckets;
};
```

- For API details, see ListBuckets in AWS SDK for JavaScript API Reference.
### Actions

#### Add CORS rules to a bucket

The following code example shows how to add cross-origin resource sharing (CORS) rules to an S3 bucket.

### SDK for JavaScript (v3)

```javascript
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.

Add a policy to a bucket

The following code example shows how to add a policy to an S3 bucket.

SDK for JavaScript (v3)

Add the policy.

```javascript
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.

Copy an object from one bucket to another

The following code example shows how to copy an S3 object from one bucket to another.
Copy the object.

```javascript
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](https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/api_reference/API_CloudFormation_CreateStack.html) in [AWS SDK for JavaScript API Reference](https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/index.html).

Create a bucket

The following code example shows how to create an S3 bucket.
Create the bucket.

```javascript
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](https://aws.amazon.com/)
- For API details, see [CreateBucket](https://aws.amazon.com/) in [AWS SDK for JavaScript API Reference](https://aws.amazon.com/)

### Delete a policy from a bucket

The following code example shows how to delete a policy from an S3 bucket.
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.

```javascript
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.

Delete an empty bucket

The following code example shows how to delete an empty S3 bucket.
Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws-samples/aws-sdk-code-examples).

Delete the bucket.

```javascript
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](https).

Delete an object

The following code example shows how to delete an S3 object.
Delete an object.

```javascript
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](#).

Delete multiple objects

The following code example shows how to delete multiple objects from an S3 bucket.
Delete multiple objects.

```javascript
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](https://aws.amazon.com/documentation/aws-sdk-for-javascript/api-reference/) in AWS SDK for JavaScript API Reference.

Delete the website configuration from a bucket

The following code example shows how to delete the website configuration from an S3 bucket.

**SDK for JavaScript (v3)**

1. **Note**
   
   There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://aws.amazon.com/code-examples/).

   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.

Get CORS rules for a bucket

The following code example shows how to get cross-origin resource sharing (CORS) rules for 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.

Get the CORS policy for the bucket.

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

const client = new S3Client({});
```javascript
export const main = async () => {
    const command = new GetBucketCorsCommand({
        Bucket: "test-bucket",
    });

    try {
        const { CORSRules } = await client.send(command);
        CORSRules.forEach((cr, i) => {
            console.log(`
        CORSRule ${i + 1}`,
        `-`.repeat(10),
        `AllowedHeaders: ${cr.AllowedHeaders.join(' ')}`,,
        `AllowedMethods: ${cr.AllowedMethods.join(' ')}`,,
        `AllowedOrigins: ${cr.AllowedOrigins.join(' ')}`,,
        `ExposeHeaders: ${cr.ExposeHeaders.join(' ')}`,,
        `MaxAgeSeconds: ${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](#).

**Get an object from a bucket**

The following code example shows how to read data from an object in 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](#).

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.

Get the ACL of a bucket

The following code example shows how to get the access control list (ACL) 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.

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.

Get the policy for a bucket

The following code example shows how to get the policy for an S3 bucket.

**SDK for JavaScript (v3)**

```javascript
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 response = await client.send(command);
        console.log(response);
    } catch (err) {
        console.error(err);
    }
};
```

**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 for a bucket

The following code example shows how to get the website configuration for an S3 bucket.

**SDK for JavaScript (v3)**

```javascript
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:`);
  }
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/).

**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws/aws-sdk-js-v3).
List the buckets.

```javascript
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'}:
    `);
  } catch (err) {
    console.error(err);
  }
};
```

For API details, see [GetBucketWebsite](https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/S3.html#GetBucketWebsite-property) in *AWS SDK for JavaScript API Reference*.

**List buckets**

The following code example shows how to list S3 buckets.

**SDK for JavaScript (v3)**

---

**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws/aws-sdk-js-v3).
List objects in a bucket

The following code example shows how to list objects in an S3 bucket.

SDK for JavaScript (v3)

```javascript
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 () => {
  try {
    let isTruncated = true;
    
    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;
      
      // Further code goes here...
    }
  }
};
```

For more information, see [AWS SDK for JavaScript Developer Guide](https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/).


There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws/aws-sdk-js-v3).
console.log("Your bucket contains the following objects:
");
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](https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/API/index.html#/aws-sdk/Client/ListObjectsV2) in **AWS SDK for JavaScript API Reference**.

**Set a new ACL for a bucket**

The following code example shows how to set a new access control list (ACL) for an S3 bucket.

**SDK for JavaScript (v3)**

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

const client = new S3Client({});
```

**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/awsdocs/aws-sdk-for-javascript-v3/tree/main/code_examples/s3).
// 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.
Set the website configuration for a bucket

The following code example shows how to set the website configuration for an S3 bucket.

SDK for JavaScript (v3)

```javascript
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);
  }
```

Set the website configuration.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws-samples/s3-set-bucket-website).
Upload an object to a bucket

The following code example shows how to upload an object to an S3 bucket.

**SDK for JavaScript (v3)**

```
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);
    }
};
```

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws-samples/aws-code-examples).
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.

```javascript
import https from "https";
import { PutObjectCommand, S3Client } from "@aws-sdk/client-s3";
import { fromIni } from "@aws-sdk/credential-providers";
import { HttpRequest } from "@aws/cryptocurrency-protocol-http";
import {
  getSignedUrl,
  S3RequestPresigner,
} from "@aws-sdk/s3-request-presigner";
import { parseUrl } from "@aws/cryptocurrency/url-parser";
import { formatUrl } from "@aws-sdk/util-format-url";
import { Hash } from "@aws/cryptocurrency/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,
            expiresIn: 3600,
        });
        console.log("Presigned URL:", noClientUrl);
    } catch (e) {
        console.error("Error creating presigned URL:", e);
    }
}
Create a presigned URL to download an object from a bucket.

```javascript
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(),
  });
  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.
const createPresignedUrlWithoutClient = ({ region, bucket, key }) => {
    const presigner = new Presigner();
    const url = `https://${bucket}.s3.amazonaws.com/${key}`;
    return formatUrl({
        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);
    }
};
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)

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.

```jsx
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](#).

**Get started with buckets and objects**

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](https://aws.amazon.com/code-examples/).

First, import all the necessary modules.

```javascript
// 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-sdk-examples/libs/utils/util-fs.js";
import {
  promptForText,
  promptToContinue,
} from "@aws-sdk-examples/libs/utils/util-io.js";
import { wrapText } from "@aws-sdk-examples/libs/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));

export const promptToSelect = (options, question = "", autoSelect) => {
  const rl = createInterface({
    input: process.stdin,
    output: process.stdout,
  });
  const selectionInvalid = (selected) =>
    isNaN(selected) || selected < 1 || selected > options.length;
  const optionsList = options.map((opt, i) => `\${i + 1}) \${opt}`).join("\n");
  const prompt = `\${question}\n\${optionsList}\n-> `;

  return new Promise((resolve) => {
    if (!selectionInvalid(autoSelect)) {
      resolve([autoSelect - 1, options[autoSelect - 1]]);
      return;
    }

    rl.question(prompt, (answer) => {
      rl.close();
      const selected = parseInt(answer);
      if (selectionInvalid(selected)) {
        console.log(
          `Invalid option. Select a number between 1 and \$\{options.length\}\`
        );
        resolve(promptToSelect(options));
      } else {
        resolve([selected - 1, options[selected - 1]]);
      }
    });
  });

export const promptToContinue = () => {
  const rl = createInterface({
    input: process.stdin,
    output: process.stdout,
  });

  return new Promise((resolve) => {
    rl.question(`
Press enter to continue.
`, () => {
      rl.close();
    });
  });
}
Objects in S3 are stored in 'buckets'. Let's define a function for creating a new bucket.

```javascript
export const createBucket = async () => {
    const bucketName = await promptForText(
        "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.

```javascript
export const uploadFilesToBucket = async ({ bucketName, folderPath }) => {
    console.log(`Uploading files from ${folderPath}\n`);
```
```javascript
const keys = readdirSync(folderPath);
const files = keys.map((key) => {
    const filePath = `${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.`);
}
```
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.

```javascript
export const downloadFilesFromBucket = async ({ bucketName }) => {
    const { Contents } = await s3Client.send(
        new ListObjectsCommand({ Bucket: bucketName }),
    );
    const path = await promptForText("Enter destination path for files:");

    for (let content of Contents) {
        const obj = await s3Client.send(
            new GetObjectCommand({ Bucket: bucketName, Key: content.Key }),
        );
    }
};
```
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.

```javascript
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.
`);
}

export const deleteBucket = async ({ bucketName }) => {
  const command = new DeleteBucketCommand({ Bucket: bucketName });
  await s3Client.send(command);
  console.log(`${bucketName} deleted successfully.
`);
};
```

The 'main' function pulls everything together. If you run this file directly the main function will be called.

```javascript
const main = async () => {
  const OBJECTDIRECTORY = `dirnameFromMetaUrl(import.meta.url,)
    .../.../.../resources/sample_files/.sample_media`;

  try {
```
```javascript
console.log(wrapText("Welcome to the Amazon S3 getting started example.")));  
console.log("Let's create a bucket.");  
const bucketName = await createBucket();  
await promptToContinue();

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 promptToContinue();

console.log(wrapText("Copy files.")));  
await copyFileFromBucket({ destinationBucket: bucketName });
await listFilesInBucket({ bucketName });
await promptToContinue();

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**
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)

```javascript
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: {
        }}),
    );
}
Download a large file.

```javascript
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}`,
  });

  await s3Client.send(command);
};
```
return s3Client.send(command);
};

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({
        Amazon S3

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 and cross-service examples.

Scenarios are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.

Each example includes a link to GitHub, where you can find instructions on how to set up and run the code in context.

Topics

• Actions

Actions

Create a vault

The following code example shows how to create an Amazon S3 Glacier vault.

SDK for JavaScript (v3)

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](https://docs.aws.amazon.com/sdk-for-javascript/v3/developerguide/)
- For API details, see [CreateVault](https://docs.aws.amazon.com/sdk-for-javascript/v3/developerguide/) in [AWS SDK for JavaScript API Reference](https://docs.aws.amazon.com/sdk-for-javascript/v3/developerguide/)

**SDK for JavaScript (v2)**

ℹ️ **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws-samples/aws-sdk-examples).
// 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.

Upload an archive to a vault

The following code example shows how to upload an archive to an Amazon S3 Glacier vault.

SDK for JavaScript (v3)

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](https://docs.aws.amazon.com/sdk-for-javascript/v3/developerguide/).
- For API details, see [UploadArchive](https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/AmazonS3.html) in [AWS SDK for JavaScript API Reference](https://docs.aws.amazon.com/sdk-for-javascript/v3/developerguide/).

**SDK for JavaScript (v2)**

ℹ️ **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws-samples/aws-s3-glacier).

// 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 and cross-service examples.

Scenarios are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.

Each example includes a link to GitHub, 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.
Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```javascript
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: ${i_NOTEBOOK_INSTANCE_NAME} Arn:${i.NotebookInstanceArn}
                           Creation Date: ${i.CreationTime.toISOString()}
                      }
                .join("\n"),
        );
    }
```

SageMaker
• For API details, see ListNotebookInstances in AWS SDK for JavaScript API Reference.

Topics
• Actions
• Scenarios

Actions

Create a pipeline

The following code example shows how to create or update a pipeline in SageMaker.

SDK for JavaScript (v3)

A function that creates a SageMaker pipeline using a locally provided JSON definition.

```javascript
/**
 * 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
 * @returns {Promise<{Arn: string}>}
 */
export async function createSagemakerPipeline({
  // Assumes an AWS IAM role has been created for this pipeline.
  roleArn,
  name,
  sagemakerClient,
});
```

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.
// 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);

const { PipelineArn } = await sagemakerClient.send(
  new CreatePipelineCommand({
    PipelineName: name,
    PipelineDefinition: pipelineDefinition,
    RoleArn: roleArn,
  })),
);
return {
  arn: PipelineArn,
  cleanUp: async () => {
    await sagemakerClient.send(
      new DeletePipelineCommand({ PipelineName: name })),
    ];
  },
};

For API details, see the following topics in AWS SDK for JavaScript API Reference.

- CreatePipeline
- UpdatePipeline

Delete a pipeline

The following code example shows how to delete a pipeline in SageMaker.
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.

```javascript
await sagemakerClient.send(
    new DeletePipelineCommand({ PipelineName: name }),
);
```

- For API details, see [DeletePipeline](#) in [AWS SDK for JavaScript API Reference](#).

### Describe a pipeline execution

The following code example shows how to describe a pipeline execution in 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](#).

Wait for a SageMaker pipeline execution to succeed, fail, or stop.

```javascript
/**
 * Poll the executing pipeline until the status is 'SUCCEEDED', 'STOPPED', or 'FAILED'.
 * @param {{ arn: string, sagemakerClient: import('@aws-sdk/client-sagemaker').SageMakerClient}} props
 */
export async function waitForPipelineComplete({ arn, sagemakerClient }) {
```
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.

Execute a pipeline

The following code example shows how to start a pipeline execution in SageMaker.
Start a SageMaker pipeline execution.

```javascript
/**
 * 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),
      },
    ],
  });
Value: JSON.stringify(jobConfig),
    ],
  },
);  
return {
  arn: PipelineExecutionArn,
};
}

- For API details, see [StartPipelineExecution](https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWSServerless/StartPipelineExecution.html) 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](https://aws.amazon.com/community/).

**SDK for JavaScript (v3)**

**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws/aws-sdk-js-v3).
The following file excerpt contains functions that use the SageMaker client to manage a pipeline.

```javascript
import { readFileSync } from "fs";

import {
    CreateRoleCommand,
    DeleteRoleCommand,
    CreatePolicyCommand,
    DeletePolicyCommand,
    AttachRolePolicyCommand,
    DetachRolePolicyCommand,
} from "@aws-sdk/client-iam";

import {
    PublishLayerVersionCommand,
    DeleteLayerVersionCommand,
    CreateFunctionCommand,
    Runtime,
    DeleteFunctionCommand,
    CreateEventSourceMappingCommand,
    DeleteEventSourceMappingCommand,
} from "@aws-sdk/client-lambda";

import {
    PutObjectCommand,
    CreateBucketCommand,
    DeleteBucketCommand,
    paginateListObjectsV2,
    DeleteObjectCommand,
    GetObjectCommand,
    ListObjectsV2Command,
} from "@aws-sdk/client-s3";

import {
    CreatePipelineCommand,
    DeletePipelineCommand,
    DescribePipelineExecutionCommand,
    PipelineExecutionStatus,
    StartPipelineExecutionCommand,
} from "@aws-sdk/client-sagemaker";
```
import { VectorEnrichmentJobDocumentType } from "@aws-sdk/client-sagemaker-geospatial";

import {
  CreateQueueCommand,
  DeleteQueueCommand,
  GetQueueAttributesCommand,
} from "@aws-sdk/client-sqs";

import { dirnameFromMetaUrl } from "@aws-sdk-examples/libs/utils/util-fs.js";
import { retry, wait } from "@aws-sdk-examples/libs/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 { Role } = await iamClient.send(
    new CreateRoleCommand({
      RoleName: name,
      AssumeRolePolicyDocument: JSON.stringify({
        Version: "2012-10-17",
        Statement: [
          {
            Effect: "Allow",
            Action: ["sts:AssumeRole"],
            Principal: { Service: ["lambda.amazonaws.com"] },
          },
        ],
      }),
    });

  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

```javascript
export async function createLambdaExecutionPolicy({
  name,
  iamClient,
  pipelineExecutionRoleArn,
}) {
  const policy = {
    Version: "2012-10-17",
    Statement: [
      {
        Effect: "Allow",
        Action: [
          "sqs:SendMessage",
          "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:
        Action: ["iam:PassRole"],
        Resource: `\${pipelineExecutionRoleArn}`,
        Condition: {
      
```
StringEquals: {
    "iam:PassedToService": [
        "sagemaker.amazonaws.com",
        "sagemaker-geospatial.amazonaws.com",
    ],
    },
},
],
};

const createPolicyCommand = new CreatePolicyCommand({
    PolicyDocument: JSON.stringify(policy),
    PolicyName: name,
});

const { Policy } = await iamClient.send(createPolicyCommand);
return {
    arn: Policy.Arn,
    policy,
    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`;

    const command = new CreateFunctionCommand({
        Code: {
            ZipFile: Uint8Array.from(readFileSync(lambdaPath)),
        },
        Runtime: Runtime.nodejs18x,
        Handler: "index.handler",
        Layers: [layerVersionArn],
        FunctionName: name,
        Role: roleArn,
    });

    // 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,
    } => lambdaClient.send(command),
    );

    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
* /

```javascript
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}} props
 * /

export async function createSagemakerRole({ name, iamClient }) {
    const command = 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", 
                        ], 
                      }, 
                    },
                    ], 
              }),
        });
```
```javascript
const { Role } = await iamClient.send(command);
// Wait for the role to be ready.
await wait(10);

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 policy = {
    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}/*`,
        ],
      },
    ],
  }
  return policy;
}
```
const createPolicyCommand = new CreatePolicyCommand({
    PolicyDocument: JSON.stringify(policy),
    PolicyName: name,
});

const { Policy } = await iamClient.send(createPolicyCommand);

return {
    arn: Policy.Arn,
    policy,
    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,)
            import.meta.url,
};
/**
 * 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 { QueueUrl } = await sqsClient.send(
    new CreateQueueCommand({
      QueueName: name,
      Attributes: {
        DelaySeconds: "5",
        ReceiveMessageWaitTimeSeconds: "5",
        VisibilityTimeout: "300",
      },
    }),
  );

  const { Attributes } = await sqsClient.send(
    new GetQueueAttributesCommand({
      QueueUrl:
    });

  const { Attributes } = await sqsClient.send(
    new GetQueueAttributesCommand({
      QueueUrl:
    });

  return {
    arn: PipelineArn,
    cleanUp: async () => {
      await sagemakerClient.send(
        new DeletePipelineCommand({ PipelineName: name }),
      );
    },
  };
}
return {
  queueUrl: QueueUrl,
  queueArn: Attributes.QueueArn,
  cleanUp: async () => {
    await sqsClient.send(new DeleteQueueCommand({ QueueUrl }));
  },
};

/**
 * Configure the AWS Lambda function to long poll for messages from the Amazon SQS
 * queue.
 * @param {{lambdaName: string, queueArn: string, lambdaClient: import('@aws-sdk/client-lambda').LambdaClient, sqsClient: import('@aws-sdk/client-sqs').SQSClient}} props
 */
export async function configureLambdaSQSEventSource({
  lambdaName,
  queueArn,
  lambdaClient,
}) {
  const { UUID } = await lambdaClient.send(
    new CreateEventSourceMappingCommand({
      EventSourceArn: queueArn,
      FunctionName: lambdaName,
    }),
  );

  return {
    cleanUp: async () => {
      await lambdaClient.send(
        new DeleteEventSourceMappingCommand({
          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.

```javascript
export async function createS3Bucket({ name, s3Client }) {
  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.

```javascript
export async function startPipelineExecution({
  sagemakerClient,
  name,
  bucketName,
  roleArn,
  s3InputBucketName,
}) {
```
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}} props
 */
export async function waitForPipelineComplete({ arn, sagemakerClient }) {
    const command = new DescribePipelineExecutionCommand({
        PipelineExecutionArn: arn,
    });

    let complete = false;
    let intervalInSeconds = 15;
    const COMPLETION_STATUSES = [
        PipelineExecutionStatus.FAILED,
        PipelineExecutionStatus.STOPPED,
    ];

    while (!complete) {
        await new Promise((resolve) => setTimeout(resolve, intervalInSeconds * 1000));
        const response = await command.send();
        complete = COMPLETION_STATUSES.includes(response.ExecutionStatus);
    }

    return { arn, executionStatus: response.ExecutionStatus, };
}
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-sdk-examples/libs/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-sdk-examples/libs/prompter.js").Prompter} prompter
* @param {import("@aws-sdk-examples/libs/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 {
    // Run all of the clean up functions. If any fail, we log the error and
    // This ensures all clean up functions are run.
    this.logger.logSeparator();
    const doCleanUp = await this.prompter.confirm({
      message: "Clean up resources?",
    });
    if (doCleanUp) {
      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,
});
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),
);

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,
    });

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({

SageMaker
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 and cross-service examples.

*Scenarios* are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.

Each example includes a link to GitHub, where you can find instructions on how to set up and run the code in context.

#### Topics
- [Actions](#)
Actions

Get a secret value

The following code example shows how to get a Secrets Manager secret value.

SDK for JavaScript (v3)

```javascript
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) [
```
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 and cross-service examples.

*Scenarios* are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.

Each example includes a link to GitHub, where you can find instructions on how to set up and run the code in context.

**Topics**

- **Actions**
Actions

Create a receipt filter

The following code example shows how to create an Amazon SES receipt filter that blocks incoming mail from an IP address or range of IP addresses.

SDK for JavaScript (v3)

```javascript
import { CreateReceiptFilterCommand, ReceiptFilterPolicy, } from '@aws-sdk/client-ses';
import { sesClient } from './libs/sesClient.js';
import { getUniqueName } from '@aws-sdk-examples/libs/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.
      },
      Name: name,
    },
  });
};
```

Note

There’s more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/awsdocs/aws-sdk-for-javascript-v3/tree/master/code-examples).
```javascript
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 (err) {
        console.log("Failed to create filter.", err);
        return err;
    }
};
```

- For API details, see [CreateReceiptFilter](https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/Ses.html) in AWS SDK for JavaScript API Reference.

**Create a receipt rule**

The following code example shows how to create an Amazon SES receipt rule.

**SDK for JavaScript (v3)**

```javascript
import { CreateReceiptRuleCommand, TlsPolicy } from "@aws-sdk/client-ses";
import { sesClient } from "./libs/sesClient.js";
import { getUniqueName } from "@aws-sdk-examples/libs/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.
Create a receipt rule set

The following code example shows how to create an Amazon SES receipt rule set to organize rules applied to incoming emails.

SDK for JavaScript (v3)

```javascript
import { CreateReceiptRuleSetCommand } from '@aws-sdk/client-ses';
import { sesClient } from './libs/sesClient.js';
import { getUniqueName } from '@aws-sdk-examples/libs/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.

Create an email template

The following code example shows how to create an Amazon SES email template.
import { CreateTemplateCommand } from '@aws-sdk/client-ses';
import { sesClient } from './libs/sesClient.js';
import { getUniqueName } from '@aws-sdk-examples/libs/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.
         *
         * The name of an existing template in Amazon SES.
         */
        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);
    }
};
For API details, see CreateTemplate in AWS SDK for JavaScript API Reference.

Delete a receipt filter

The following code example shows how to delete an Amazon SES receipt filter.

SDK for JavaScript (v3)

```javascript
import { DeleteReceiptFilterCommand } from '@aws-sdk/client-ses';
import { sesClient } from './libs/sesClient.js';
import { getUniqueName } from '@aws-sdk-examples/libs/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;
  }
};
```

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 API details, see `DeleteReceiptFilter` in *AWS SDK for JavaScript API Reference*.

### Delete a receipt rule

The following code example shows how to delete an Amazon SES receipt rule.

**SDK for JavaScript (v3)**

```javascript
import { DeleteReceiptRuleCommand } from '@aws-sdk/client-ses';
import { getUniqueName } from '@aws-sdk-examples/libs/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;
    }
};
```

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws-samples/aws-sdk-code-examples).
Delete a rule set

The following code example shows how to delete an Amazon SES rule set and all of the rules it contains.

**SDK for JavaScript (v3)**

```javascript
import { DeleteReceiptRuleSetCommand } from '@aws-sdk/client-ses';
import { getUniqueName } from '@aws-sdk-examples/libs/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](https://aws.amazon.com/documentation/sdk-for-javascript/api-reference/) in AWS SDK for JavaScript API Reference.
Delete an email template

The following code example shows how to delete an Amazon SES email template.

**SDK for JavaScript (v3)**

```javascript
import { DeleteTemplateCommand } from '@aws-sdk/client-ses';
import { getUniqueName } from '@aws-sdk-examples/libs/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*.

Delete an identity

The following code example shows how to delete an Amazon SES identity.
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.

Get an existing email template

The following code example shows how to get an existing Amazon SES email template.
import { GetTemplateCommand } from '@aws-sdk/client-ses';
import { getUniqueName } from '@aws-sdk-examples/libs/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 (err) {
    console.log("Failed to get email template.", err);
    return err;
  }
};

• For API details, see GetTemplate in AWS SDK for JavaScript API Reference.

List email templates

The following code example shows how to list Amazon SES email templates.
SDK for JavaScript (v3)

Note
There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```javascript
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.

List identities

The following code example shows how to list Amazon SES identities.

SDK for JavaScript (v3)

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.

List receipt filters

The following code example shows how to list Amazon SES receipt filters.

SDK for JavaScript (v3)

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);
};

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 bulk templated email

The following code example shows how to send templated email to multiple destinations with Amazon SES.

SDK for JavaScript (v3)

```javascript
import { SendBulkTemplatedEmailCommand } from '@aws-sdk/client-ses';
import {
    getUniqueName,
    postfix,
} from '@aws-sdk-examples/libs/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 },
];
```

- For API details, see [ListReceiptFilters](#) in *AWS SDK for JavaScript API Reference*.
/**
 * @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 (err) {
    console.log("Failed to send bulk template email", err);
    return err;
  }
};
For API details, see `SendBulkTemplatedEmail` in *AWS SDK for JavaScript API Reference*.

**Send email**

The following code example shows how to send email with Amazon SES.

### SDK for JavaScript (v3)

```javascript
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",
        },
      },
    },
  });
};
```

---

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws-samples/aws-sdk-javascript-get-started).

---

Amazon SES
• For API details, see [SendEmail](https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/ses-send-email.html) in AWS SDK for JavaScript API Reference.

**Send raw email**

The following code example shows how to send a raw email with Amazon SES.
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.

```javascript
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);
        }
      }
    );
  });
```
import { SendTemplatedEmailCommand } from '@aws-sdk/client-ses';
import {
  getUniqueName,
  postfix,
} from '@aws-sdk-examples/libs/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");

Send templated email

The following code example shows how to send templated email with Amazon SES.

SDK for JavaScript (v3)

For API details, see SendRawEmail in AWS SDK for JavaScript API Reference.
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 (err) {
        console.log("Failed to send template email", err);
        return err;
    }
};

• For API details, see SendTemplatedEmail in AWS SDK for JavaScript API Reference.

Update an email template

The following code example shows how to update an Amazon SES email template.
Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```javascript
import { UpdateTemplateCommand } from '@aws-sdk/client-ses';
import { getUniqueName } from '@aws-sdk-examples/libs/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.
Verify a domain identity

The following code example shows how to verify a domain identity with Amazon SES.

SDK for JavaScript (v3)

```javascript
import { VerifyDomainIdentityCommand } from '@aws-sdk/client-ses';
import {
    getUniqueName,
    postfix,
} from '@aws-sdk-examples/libs/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](https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/SES.html#VerifyDomainIdentityCommand) in [AWS SDK for JavaScript API Reference](https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/SES.html).

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/awsdocs/aws-sdk-js-examples/tree/master/examples/email).
Verify an email identity

The following code example shows how to verify an email identity with Amazon SES.

SDK for JavaScript (v3)

```javascript
// 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;
    }
};
```


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 and cross-service examples.

Scenarios are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.

Each example includes a link to GitHub, 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)

```javascript
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) {
```

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 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

Actions

Check whether a phone number is opted out

The following code example shows how to check whether a phone number is opted out of receiving Amazon SNS messages.

SDK for JavaScript (v3)

ℹ️ 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.

```javascript
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);

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [CheckIfPhoneNumberIsOptedOut](#) in [AWS SDK for JavaScript API Reference](#).

Confirm an endpoint owner wants to receive messages

The following code example shows how to confirm the owner of an endpoint wants to receive Amazon SNS messages by validating the token sent to the endpoint by an earlier Subscribe action.

Amazon SNS
Create the client in a separate module and export it.

```javascript
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.

```javascript
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(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:xxxxxxxxxxxx-
//   xxxx-xxxx-xxxx-xxxxxxxxxxxx'
// }
return response;

- For more information, see AWS SDK for JavaScript Developer Guide.
- For API details, see ConfirmSubscription in AWS SDK for JavaScript API Reference.

Create a topic

The following code example shows how to create an Amazon SNS topic.

SDK for JavaScript (v3)

```javascript
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.
```

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 { 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
    //     },
    // }
    return response;
};

• For more information, see AWS SDK for JavaScript Developer Guide.
• For API details, see CreateTopic in AWS SDK for JavaScript API Reference.

Delete a subscription

The following code example shows how to delete an Amazon SNS subscription.
Create the client in a separate module and export it.

```javascript
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.

```javascript
import { UnsubscribeCommand } from '@aws-sdk/client-sns';
import { snsClient } from '../libs/snsClient.js';

/**
 * @param {string} subscriptionArn - The ARN of the subscription to cancel.
 */
  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, 
  //   }, 
  //   'Message': '' 
  // } 
}
```
Delete a topic

The following code example shows how to delete an Amazon SNS topic and all subscriptions to that topic.

SDK for JavaScript (v3)

Create the client in a separate module and export it.

```javascript
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.

```javascript
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](https://aws.amazon.com/sdk-for-javascript/getting-started/).

Get the properties of a topic

The following code example shows how to get the properties of an Amazon SNS topic.

SDK for JavaScript (v3)

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws-examples/)

Create the client in a separate module and export it.

```javascript
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.

```javascript
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": 'xxxxxxxxxxxx',
    //     "SubscriptionsPending": '1',
    //     "TracingConfig": 'PassThrough',
    //     "EffectiveDeliveryPolicy": '{"http":{"defaultHealthyRetryPolicy":
    //       "minDelayTarget":20,"maxDelayTarget":20,"numRetries":3,"numMaxDelayRetries":0,"numNoDelayRetries":0,"numMinDelayRetries":0,"backoffFunction":"linear"},"disableSubscriptionOverrides":false,"defaultRequestPolicy":
    //       "headerContentType":"text/plain; charset=UTF-8"}}',
    //     "SubscriptionsConfirmed": '0',
    //     "DisplayName": '',
    //     "SubscriptionsDeleted": '1'
    //   }
    // };
    return response;
};
```
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](https://github.com/aws/aws-sdk-js-examples).

Import the SDK and client modules and call the API.

```javascript
// 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](https).

### Get the settings for sending SMS messages

The following code example shows how to get the settings for sending Amazon SNS SMS messages.
Create the client in a separate module and export it.

```javascript
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.

```javascript
import { GetSMSAttributesCommand } from '@aws-sdk/client-sns';
import { snsClient } from '../libs/snsClient.js';

export const getSmsAttributes = async () => {
  const response = await snsClient.send(
    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' }
  // }
```
List the subscribers of a topic

The following code example shows how to retrieve the list of subscribers of an Amazon SNS topic.

SDK for JavaScript (v3)

```javascript
// 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 { 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_ARNAObama') => {
  // }
  return response;
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](https://aws.amazon.com/sdk/js/).
- For API details, see [GetSMSAttributes](https://docs.aws.amazon.com/AWSSDK/latest/APIReference/API_GetSMSAttributes.html) in [AWS SDK for JavaScript API Reference](https://docs.aws.amazon.com/AWSSDK/latest/APIReference/).
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',
//     }
//   ]
// },
// return response;
};

• For more information, see AWS SDK for JavaScript Developer Guide.
• For API details, see ListSubscriptions in AWS SDK for JavaScript API Reference.

List topics

The following code example shows how to list Amazon SNS topics.

SDK for JavaScript (v3)

ℹ️ 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.

```javascript
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.

```javascript
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:xxxxxxxxxxx: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 a message with an attribute

The following code example shows how to publish a message with an attribute using Amazon SNS.
Publish a message to a topic with group, duplication, and attribute options.

```javascript
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,
                }
            )
        })
    );
}
```
const publishAnother = await this.prompter.confirm({
    message: MESSAGES.publishAnother,
  });

if (publishAnother) {
    await this.publishMessages();
}

• For API details, see Publish in AWS SDK for JavaScript API Reference.

Publish to a topic

The following code example shows how to publish messages to an Amazon SNS topic.

SDK for JavaScript (v3)

⚠️ 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.

```javascript
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.

```javascript
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;
};

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [Publish](#) in [AWS SDK for JavaScript API Reference](#).

Set the default settings for sending SMS messages

The following code example shows how to set the default settings for sending SMS messages 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](#).

Create the client in a separate module and export it.

```javascript
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.

```javascript
import { SetSMSAttributesCommand } from '@aws-sdk/client-sns';
import { snsClient } from '../libs/snsClient.js';

/**
 * @param {"Transactional" | "Promotional"} defaultSmsType
 */
export const setSmsType = async (defaultSmsType = "Transactional") => {
```
```javascript
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](https://aws.amazon.com/documentation/js-sdk/).

**Set topic attributes**

The following code example shows how to set Amazon SNS topic attributes.

**SDK for JavaScript (v3)**

ℹ️ **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws/aws-sdk-js-v3).

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 { 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 a Lambda function to a topic

The following code example shows how to subscribe a Lambda function so it receives notifications from an Amazon SNS topic.

SDK for JavaScript (v3)

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.

```javascript
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.

```javascript
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,
    }));
```
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.

Subscribe a mobile application to a topic

The following code example shows how to subscribe a mobile application endpoint so it receives notifications from an Amazon SNS topic.

SDK for JavaScript (v3)

Create the client in a separate module and export it.

For more information, see [AWS SDK for JavaScript Developer Guide](https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/).


Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws/aws-sdk-js-v3).
export const snsClient = new SNSClient({});

Import the SDK and client modules and call the API.

```javascript
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-b3c0-5940-9f66-06f6fccc108f0',
  //     extendedRequestId: undefined,
  //     cfId: undefined,
  //     attempts: 1,
  //     totalRetryDelay: 0
  //   },
  //   SubscriptionArn: 'pending confirmation'
  // };
  return response;
};
```

- For more information, see [AWS SDK for JavaScript Developer Guide](#).
- For API details, see [Subscribe](#) in [AWS SDK for JavaScript API Reference](#).
Subscribe an SQS queue to a topic

The following code example shows how to subscribe an Amazon SQS queue so it receives notifications from an Amazon SNS topic.

SDK for JavaScript (v3)

```javascript
import { SubscribeCommand, SNSClient } from '@aws-sdk/client-sns';

const client = new SNSClient({});

export const subscribeQueue = async (topicArn = 'TOPIC_ARН', queueArn = 'QUEUE_ARН') => {
    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:xxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx'
    // }
    return response;
```

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.
Subscribe an email address to a topic

The following code example shows how to subscribe an email address to an Amazon SNS topic.

SDK for JavaScript (v3)

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.

```javascript
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.

```javascript
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-b3c0-5940-9f66-06f6fccc108f0',
//      extendedRequestId: undefined,
//      cfId: undefined,
//      attempts: 1,
//      totalRetryDelay: 0
//   },
//   SubscriptionArn: 'pending confirmation'
// };

- For more information, see AWS SDK for JavaScript Developer Guide.
- For API details, see Subscribe in AWS SDK for JavaScript API Reference.

Subscribe with a filter to a topic

The following code example shows how to subscribe with a filter to an Amazon SNS topic.

SDK for JavaScript (v3)

Note
There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```javascript
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
  //   },
  // }
  return response;
};

• For API details, see Subscribe in AWS SDK for JavaScript API Reference.

Scenarios

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.

```javascript
import { SNSClient } from '@aws-sdk/client-sns';
import { SQSClient } from '@aws-sdk/client-sqs';
import { TopicsQueuesWkflw } from './TopicsQueuesWkflw.js';
import { Prompter } from '@aws-sdk-examples/libs/prompter.js';
import { SlowLogger } from '@aws-sdk-examples/libs/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() {
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)
        )
    }
}
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) {
        await this.snsClient.receiveMessage({
            QueueUrl: queue,
            MAX_NUMBER_OF_MESSAGES: 10,
            WAIT_TIME_SECONDS: 20,
        });
        await this.snsClient.deleteMessage({
            QueueUrl: queue,
            ReceiptHandle: receiptHandle,
        });
    }
}
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](#)

**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 and cross-service examples.

*Scenarios* are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.

Each example includes a link to GitHub, 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.
Initialize an Amazon SQS client and list queues.

```javascript
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('
'));
};
```

• For API details, see [ListQueues](#) in AWS SDK for JavaScript API Reference.

**Topics**

Amazon SQS
Actions

Change message timeout visibility

The following code example shows how to change an Amazon SQS message timeout visibility.

SDK for JavaScript (v3)

```javascript
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);
};
```

Receive an Amazon SQS message and change its timeout visibility.
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.

```javascript
// 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.

Configure a dead-letter queue

The following code example shows how to configure a dead-letter queue in Amazon SQS.

SDK for JavaScript (v3)

```javascript
import { SetQueueAttributesCommand, SQSClient } from '@aws-sdk/client-sqs';
```

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 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*.

**Create a queue**

The following code example shows how to create an Amazon SQS queue.

**SDK for JavaScript (v3)**

ℹ️ **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.

```javascript
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.

```javascript
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.
                ReceiveMessageWaitTimeSeconds: '20',
            },
        })),
    
    return response;
};
```
Create an Amazon SQS standard queue.

```javascript
// 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);
  }
});
```

- For more information, see [AWS SDK for JavaScript Developer Guide](https://docs.aws.amazon.com/sdk-for-javascript/latest/developer-guide/).
- For API details, see [CreateQueue](https://docs.aws.amazon.com/sdk-for-javascript/latest/developer-guide/aws-sdk-api-sqs-createqueue.html) in [AWS SDK for JavaScript API Reference](https://docs.aws.amazon.com/sdk-for-javascript/latest/developer-guide/).

**SDK for JavaScript (v2)**

ℹ️ **Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws/aws-sdk-js-v3/tree/master/example).
Create an Amazon SQS queue that waits for a message to arrive.

```javascript
// 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](https://docs.aws.amazon.com/sdk-for-javascript/v3/developerguide/).
- For API details, see [CreateQueue](https://docs.aws.amazon.com/AWSJavaScriptSDK/v3/docs/interfaces/sqsqueue.html#createqueue-property) in [AWS SDK for JavaScript API Reference](https://docs.aws.amazon.com/AWSJavaScriptSDK/v3/).

Delete a batch of messages from a queue

The following code example shows how to delete a batch of messages 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,
            }));
    }
}
Receive and delete Amazon SQS messages.

```javascript
import {
  ReceiveMessageCommand,
  DeleteMessageCommand,
  SQSClient,
  DeleteMessageBatchCommand,
} from '@aws-sdk/client-sqs';

const client = new SQSClient({});
```

For API details, see [DeleteMessageBatch](https://aws-sdk.github.io/aws-sdk-js-v3/API_reference.html) in *AWS SDK for JavaScript API Reference*.

### Delete a message from a queue

The following code example shows how to delete a message from an Amazon SQS queue.

**SDK for JavaScript (v3)**

```javascript
await client.send(
  new DeleteMessageBatchCommand({
    QueueUrl: queueUrl,
    Entries: Messages.map((message) => ({
      Id: message.MessageId,
      ReceiptHandle: message.ReceiptHandle,
    })),
  }));
}
```

---

- Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/awsdocs/aws-sdk-js-v3).
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](https://aws.amazon.com/sdk-for-javascript/api/latest/) in *AWS SDK for JavaScript API Reference*. 
Receive and delete Amazon SQS messages.

```javascript
// 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) {
    ```
Delete a queue

The following code example shows how to delete an Amazon SQS queue.

SDK for JavaScript (v3)

```javascript
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 [DeleteMessage](#) in [AWS SDK for JavaScript API Reference](#).

• 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)**

ℹ️ **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.

```javascript
// 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*.

**Get attributes for a queue**

The following code example shows how to get attributes for an Amazon SQS queue.
import { GetQueueAttributesCommand, SQSClient } from '@aws-sdk/client-sqs';

const client = new SQSClient({});
const SQS_QUEUE_URL = 'queue-url';

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.

Get the URL of a queue

The following code example shows how to get the URL of an Amazon SQS queue.
Get the URL for an Amazon SQS queue.

```javascript
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.

Get the URL for an Amazon SQS queue.

```javascript
// 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.

List queues

The following code example shows how to list Amazon SQS queues.

SDK for JavaScript (v3)

ℹ️ 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.

```javascript
import { paginateListQueues, SQSClient } from "@aws-sdk/client-sqs";

const client = new SQSClient({});

export const main = async () => {
  const paginatedListQueues = paginateListQueues({ client }, {});
```
```javascript
/** @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](https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/).
- For API details, see [ListQueues](https://docs.aws.amazon.com/AWSJavaScriptSDK/v3/docs/api-reference/) in [AWS SDK for JavaScript API Reference](https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/).

**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](https://github.com/aws-examples).

List your Amazon SQS queues.

```javascript
// 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);
  }
});
```
Receive messages from a queue

The following code example shows how to receive messages from an Amazon SQS queue.

**SDK for JavaScript (v3)**

```javascript
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,  
    }),
  )
```

**Note**

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws-samples/).
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

const command = { 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: [
Send a message to a queue

The following code example shows how to send a message to an Amazon SQS queue.

SDK for JavaScript (v3)

```
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,
    });
    sqs.sendMessage(command, function(err, data) {
        if (err) {
            console.log("Error", err);
        } else {
            console.log("Success", data);
        }
    });
}
```

- For more information, see [AWS SDK for JavaScript Developer Guide](https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/).

---

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws-samples/aws-code-examples).
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.

Set queue attributes

The following code example shows how to set the attributes for an Amazon SQS queue.
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;
};
```javascript
const response = await client.send(command);
console.log(response);
return response;
```

- For API details, see [SetQueueAttributes](#) in [AWS SDK for JavaScript API Reference](#).

## Scenarios

### 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)

```javascript
import { SNSClient } from '@aws-sdk/client-sns';
import { SQSClient } from '@aws-sdk/client-sqs';
import { TopicsQueuesWkflw } from './TopicsQueuesWkflw.js';
import { Prompter } from '@aws-sdk-examples/libs/prompter.js';
import { SlowLogger } from '@aws-sdk-examples/libs/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.autoDedupe ? { 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,
    )
async subscribeQueuesToTopic() {
  for (const [index, queue] of this.queues.entries()) {
    /**
     * @type {import('@aws-sdk/client-sns').SubscribeCommandInput}
     */
    const subscribeParams = {
      TopicArn: this.topicArn,
      Protocol: "sqs",
      // Other parameters
    };
    try {
      await this.sqsClient.subscribe(subscribeParams);
      queue.subscriptions.push(subscribeParams);
    } catch (error) {
      console.error(error);
    }
  }
}
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),
                    },
                },
            : {})
    });
AWS SDK for JavaScript

Developer Guide for SDK Version 3

},
}
: {}),
}),
);
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(

Amazon SQS

978


"${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();
  } catch (err) {
    console.error(err);
  }
}
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](#)
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 and cross-service examples.

*Scenarios* are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.

Each example includes a link to GitHub, where you can find instructions on how to set up and run the code in context.

**Topics**

- *Actions*

**Actions**

**Assume a role**

The following code example shows how to assume a role with AWS STS.

**SDK for JavaScript (v3)**

*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.

```javascript
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.

```javascript
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](https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWSSDK.html) in [AWS SDK for JavaScript API Reference](https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWSSDK.html).

**SDK for JavaScript (v2)**

ℹ️ **Note**
There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/aws/aws-sdk-js-v3).
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.

*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 and cross-service examples.

*Scenarios* are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.

Each example includes a link to GitHub, 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](https://github.com/aws/aws-sdk-js-v3).

---

Invoke `main()` to run the example.

```javascript
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 () => {
    // Your code here
}
```
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
• Actions
• Scenarios

Actions

Add a communication to a case

The following code example shows how to add an AWS Support communication with an attachment to a support case.
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*.

### Add an attachment to a set

The following code example shows how to add an AWS Support attachment to an attachment set.
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);
  }
};
Create a case

The following code example shows how to create a new AWS Support case.

SDK for JavaScript (v3)

```javascript
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);
  }
};
```

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.
Describe an attachment

The following code example shows how to describe an attachment for an AWS Support case.

SDK for JavaScript (v3)

```javascript
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 CreateCase in AWS SDK for JavaScript API Reference.

For API details, see DescribeAttachment 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.
Describe cases

The following code example shows how to describe AWS Support cases.

SDK for JavaScript (v3)

```javascript
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/describe-cases-command-input.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.

Describe communications

The following code example shows how to describe AWS Support communications for a case.
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.
    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.

Describe severity levels

The following code example shows how to describe AWS Support severity levels.
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.

Resolve case

The following code example shows how to resolve an AWS Support case.
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.

Scenarios

Get started with cases

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.
Run an interactive scenario in the terminal.

```javascript
import {
    AddAttachmentsToSetCommand,
    AddCommunicationToCaseCommand,
    CreateCaseCommand,
    DescribeAttachmentCommand,
    DescribeCasesCommand,
    DescribeCommunicationsCommand,
    DescribeServicesCommand,
    DescribeSeverityLevelsCommand,
    ResolveCaseCommand,
    SupportClient,
} from '@aws-sdk/client-support';
import inquirer from 'inquirer';

// Retry an asynchronous function on failure.
const retry = async ({ intervalInMs = 500, maxRetries = 10 }, fn) => {
    try {
        return await fn();
    } catch (err) {
        console.log('Function call failed. Retrying.');
        console.error(err.message);
        if (maxRetries === 0) throw err;
        await new Promise((resolve) => setTimeout(resolve, intervalInMs));
        return retry({ intervalInMs, maxRetries: maxRetries - 1 }, fn);
    }
};

const wrapText = (text, char = "=") => {
    const rule = char.repeat(80);
    return `${rule}
    ${text}
${rule}
`;
};

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;
        }
    }
}

// Get the list of available services.
export const getService = async () => {
    const { services } = await client.send(new DescribeServicesCommand({}));
    const { selectedService } = await inquirer.prompt({
        name: "selectedService",
        type: "list",
        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;
}

// Get the list of available support case categories for a service.
export const getCategory = async (service) => {
    const { selectedCategory } = await inquirer.prompt({
        name: "selectedCategory",
        type: "list",
        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.prompt({
        name: "selectedSeverityLevel",
        type: "list",
        message: "Select a severity level.",
        choices: severityLevels.map((s) => ({ name: s.name, value: s })),
    });
    return selectedSeverityLevel;
};

// Create a new support case and return the caseId.
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." spokes
    );
}
// 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;
};

// Get an attachment set.
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.prompt({
        name: "shouldResolve",
        type: "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.
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(`\nOpen support cases created today: ${todaysOpenCases.length}`);

// 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: ' +
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(
    '
Waiting for case status to be marked as resolved. This can take some time.'
  );
  const resolvedCases = await retry(
    { intervalInMs: 20000, maxRetries: 15 },
    () => getTodaysResolvedCases(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
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 and cross-service examples.

Scenarios are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.

Each example includes a link to GitHub, where you can find instructions on how to set up and run the code in context.

Topics
• Actions

Actions

Delete a medical transcription job

The following code example shows how to delete an Amazon Transcribe Medical transcription job.

SDK for JavaScript (v3)

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.

```javascript
const { TranscribeClient } = require("@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 });
```
```javascript
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](https://aws.amazon.com/sdk-for-javascript/).
- For API details, see [DeleteMedicalTranscriptionJob](https://aws.amazon.com/sdk-for-javascript/api-reference/) in [AWS SDK for JavaScript API Reference](https://aws.amazon.com/sdk-for-javascript/api-reference/).

Delete a transcription job

The following code example shows how to delete an Amazon Transcribe transcription job.
Delete a transcription job.

```javascript
// 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.

```javascript
const { TranscribeClient } = require('@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

The following code example shows how to list Amazon Transcribe Medical transcription jobs.

SDK for JavaScript (v3)

```javascript
// 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'
}
```

Create the client.

```javascript
const { TranscribeClient } = require('@aws-sdk/client-transcribe');
const REGION = "REGION"; // e.g. "us-east-1"
const transcribeClient = new TranscribeClient({ region: REGION });
export { transcribeClient };
```
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.

List transcription jobs

The following code example shows how to list Amazon Transcribe transcription jobs.

SDK for JavaScript (v3)

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 { 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.

const { TranscribeClient } = require("@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 };
Create the client.

```javascript
const { TranscribeClient } = require('@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.

```javascript
// 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 () => {
    // Code to start the transcription job
};
```
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](https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/).

### Start a transcription job

The following code example shows how to start an Amazon Transcribe transcription job.

#### SDK for JavaScript (v3)

```javascript
// 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.

const { TranscribeClient } = require("@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 };
Examples

- **Build an Amazon Transcribe app**
- **Build an Amazon Transcribe streaming app**
- **Build an application to submit data to a DynamoDB table**
- **Create an Amazon Lex chatbot to engage your website visitors**
- **Create a photo asset management application that lets users manage photos using labels**
- **Create a web application to track DynamoDB data**
- **Create an Aurora Serverless work item tracker**
- **Create an Amazon Textract explorer application**
- **Create an application that analyzes customer feedback and synthesizes audio**
- **Detect PPE in images with Amazon Rekognition using an AWS SDK**
- **Detect objects in images with Amazon Rekognition using an AWS SDK**
- **Detect people and objects in a video with Amazon Rekognition using an AWS SDK**
- **Invoke a Lambda function from a browser**
- **Use API Gateway to invoke a Lambda function**
- **Use Step Functions to invoke Lambda functions**
- **Use scheduled events to invoke a Lambda function**

**Build an Amazon Transcribe app**

**SDK for JavaScript (v3)**

Create an app that uses Amazon Transcribe to transcribe and display voice recordings in the browser. The app uses two Amazon Simple Storage Service (Amazon S3) buckets, one to host the application code, and another to store transcriptions. The app uses an Amazon Cognito user pool to authenticate your users. Authenticated users have AWS Identity and Access Management (IAM) permissions to access the required AWS services.

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](https://aws.amazon.com/).
Services used in this example

- Amazon Cognito Identity
- Amazon S3
- Amazon Transcribe

Build an Amazon Transcribe streaming app

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

Build an application to submit data to a DynamoDB 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 an Amazon Lex 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 website 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 a photo asset management 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

• DynamoDB
• Lambda
• Amazon Rekognition
• Amazon S3
Create a web application to track DynamoDB data

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

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
Create an Amazon Textract explorer 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 that analyzes customer feedback and synthesizes audio

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.

```javascript
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,
    }
```

Create an application to analyze customer feedback
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(" ");
};
/**
 * 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";
/**
Create an application to analyze customer feedback
1018
* 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

Detect PPE in images with Amazon Rekognition using an AWS SDK

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 with Amazon Rekognition using an AWS SDK

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
Detect people and objects in a video with Amazon Rekognition using an AWS SDK

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

Invoke a 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.

Use API Gateway to invoke a Lambda function

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 Step Functions to invoke Lambda functions

SDK for JavaScript (v3)

Shows how to create an AWS serverless workflow by using AWS Step Functions and the AWS SDK for JavaScript. Each workflow step is implemented using an AWS Lambda function.

Lambda is a compute service that enables you to run code without provisioning or managing servers. Step Functions is a serverless orchestration service that lets you combine Lambda functions and other AWS services to build business-critical applications.

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
- Lambda
- Amazon SES
- Step Functions

Use scheduled events to invoke a Lambda function

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
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
- Enforcing 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 140-2 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) 140-2.

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 <a href="https://docs.aws.amazon.com/sections/how-to-sign-in-to-your-aws-account">How to sign in to your AWS account</a> in the <i>AWS Sign-In User Guide</i>.

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 <a href="https://docs.aws.amazon.com/sections/signing-aws-api-requests">Signing AWS API requests</a> in the <i>IAM User Guide</i>.

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 <a href="https://docs.aws.amazon.com/sections/multifactor-authentication">Multi-factor authentication</a> in the <i>AWS IAM Identity Center User Guide</i> and <a href="https://docs.aws.amazon.com/sections/using-multi-factor-authentication-mfa">Using multi-factor authentication (MFA) in AWS</a> in the <i>IAM User Guide</i>.

**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 <i>root user</i> 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 <a href="https://docs.aws.amazon.com/sections/tasks-that-require-root-user-credentials">Tasks that require root user credentials</a> in the <i>IAM User Guide</i>.

**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 <i>federated identity</i> 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 permissions 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 *How IAM roles differ from resource-based policies* 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 How SCPs work 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 How IAM roles differ from resource-based policies 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.
AWS SDK for JavaScript

Developer Guide for SDK Version 3

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](#).

- **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](https://aws.amazon.com/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](https://docs.aws.amazon.com/security/security-documentation) and [AWS services that are in scope of AWS compliance efforts by compliance program](https://aws.amazon.com/compliance/).

**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](https://docs.aws.amazon.com/security/security-documentation). To design your AWS environment using the best practices for infrastructure security, see [Infrastructure Protection](https://docs.aws.amazon.com/security/security-documentation) in [Security Pillar AWS Well-Architected Framework](https://aws.amazon.com/wellarchitected/

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](https://docs.aws.amazon.com/AmazonSTS/latest/APIReference/index.html) (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](https://docs.aws.amazon.com/security/security-documentation) and [AWS services that are in scope of AWS compliance efforts by compliance program](https://aws.amazon.com/compliance/).
Enforcing 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.

```bash
node -p process.versions
```

The version of OpenSSL in the list is the version used by Node.js, as shown in the following example.

```bash
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.
```javascript
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
```

**Enforcing 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.

```javascript
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'
    })
  })
});

Enforcing 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.
Document history for AWS SDK for JavaScript version 3

Document History

- **Latest documentation update:** August 22, 2022

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](#).

<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Announcement</strong></td>
<td>Updated top banner with an end-of-support reminder for Internet Explorer 11.</td>
<td>September 23, 2022</td>
</tr>
<tr>
<td><strong>Minor updates</strong></td>
<td>Minor updates to clarity and resolving broken links. Added awareness links to AWS SDKs and Tools Reference Guide.</td>
<td>August 22, 2022</td>
</tr>
<tr>
<td><strong>Enforcing a minimum TLS version</strong></td>
<td>Added information about TLS 1.3.</td>
<td>March 31, 2022</td>
</tr>
<tr>
<td><strong>Updated AWS Lambda tutorial</strong></td>
<td>Added tutorial demonstrating how to build a browser-based application for submitting data to a Amazon DynamoDB table.</td>
<td>October 20, 2020</td>
</tr>
<tr>
<td><strong>Setting credentials in Node.js topic updated</strong></td>
<td>Update topic about setting credentials in Node.js for AWS SDK for JavaScript V3.</td>
<td>October 20, 2020</td>
</tr>
<tr>
<td><strong>Migrating to V3</strong></td>
<td>Added topic to describe how to migrate to AWS SDK for JavaScript V3.</td>
<td>October 20, 2020</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
<td>Date</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td><strong>Getting Started</strong></td>
<td>Updated topics for getting started in the browser and getting started with Node.js for AWS SDK for JavaScript V3.</td>
<td>October 20, 2020</td>
</tr>
<tr>
<td><strong>Browser builder</strong></td>
<td>Information about AWS Browser Builder was removed because it is not required for AWS SDK for JavaScript V3.</td>
<td>October 20, 2020</td>
</tr>
<tr>
<td><strong>Amazon Transcribe service examples updated</strong></td>
<td>Updated Amazon Transcribe service examples for AWS SDK for JavaScript V3.</td>
<td>October 20, 2020</td>
</tr>
<tr>
<td><strong>Amazon Simple Notification Service examples updated</strong></td>
<td>Updated Amazon Simple Notification Service service examples for AWS SDK for JavaScript V3.</td>
<td>October 20, 2020</td>
</tr>
<tr>
<td><strong>Amazon Simple Email Service examples updated</strong></td>
<td>Updated Amazon Simple Email Service service examples for AWS SDK for JavaScript V3.</td>
<td>October 20, 2020</td>
</tr>
<tr>
<td><strong>Amazon Redshift service examples updated</strong></td>
<td>Updated Amazon Redshift service examples for AWS SDK for JavaScript V3.</td>
<td>October 20, 2020</td>
</tr>
<tr>
<td><strong>Amazon Lex service examples updated</strong></td>
<td>Updated Amazon Lex service examples for AWS SDK for JavaScript V3.</td>
<td>October 20, 2020</td>
</tr>
<tr>
<td><strong>Amazon DynamoDB service examples updated</strong></td>
<td>Updated Amazon DynamoDB service examples for AWS SDK for JavaScript V3.</td>
<td>October 20, 2020</td>
</tr>
<tr>
<td>Service</td>
<td>Description</td>
<td>Date</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>AWS Elemental MediaConvert</td>
<td>Updated AWS Elemental MediaConvert service examples for AWS SDK for JavaScript V3.</td>
<td>October 20, 2020</td>
</tr>
<tr>
<td>AWS Lambda service</td>
<td>Updated AWS Lambda service examples for AWS SDK for JavaScript V3.</td>
<td>October 20, 2020</td>
</tr>
<tr>
<td>AWS SDK for JavaScript V3</td>
<td>Released pre-release version of the AWS SDK for JavaScript V3 Developer Guide.</td>
<td>October 19, 2020</td>
</tr>
</tbody>
</table>