AWS SDK for JavaScript

Developer Guide for SDK Version 3

AWS SDK for JavaScript: Developer Guide for SDK Version 3

Table of Contents

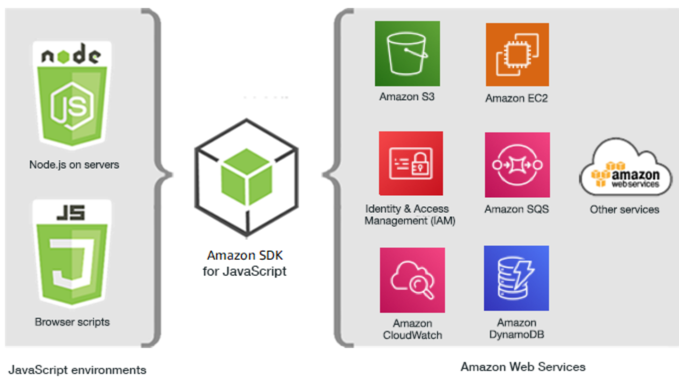
Help us improve the AWS SDK for JavaScript version 3 (V3) documentation by providing feedback using the **Feedback** link, or create an issue or pull request on GitHub.

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 (p. 56).

Modularized packages

Version 2 of the SDK for JavaScript (V2) required you to use the entire AWS SDK, as follows.

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

Loading the entire SDK isn't an issue if your application is using many AWS services. However, if you need to use only a few AWS services, it means increasing the size of your application with code you don't need or use.

In V3, you can load and use only the individual AWS Services you need. This is shown in the following example, which gives you access to Amazon DynamoDB (DynamoDB).

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

Not only can you load and use individual AWS services, but you can also load and use only the service commands you need. This is shown in the following examples, which gives you access to DynamoDB client and the `ListTablesCommand` command.

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

Important
You should not import submodules into modules. For example, the following code might result in errors.

```
import {CognitoIdentity} from "@aws-sdk/client-cognito-identity/CognitoIdentity";
```

The following is the correct code.

```
import {CognitoIdentity} from "@aws-sdk/client-cognito-identity";
```

Comparing code size

In Version 2 (V2), a simple code example that lists all of your Amazon DynamoDB tables in the `us-west-2` Region might look like the following.

```
var AWS = require("aws-sdk");
// Set the Region
AWS.config.update({region: "us-west-2"});
// Create DynamoDB service object
var ddb = new AWS.DynamoDB({apiVersion: "2006-03-01"});

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

```
  }
});
```

V3 looks like the following.

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

  try {
    const results = await dbclient.send(new ListTablesCommand);
    results.Tables.forEach(function (item, index) {
      console.log(item.Name);
    });
  } catch (err) {
    console.error(err)
  }
})();
```

The `aws-sdk` package adds about 40 MB to your application. Replacing `var AWS = require("aws-sdk")` with `import {DynamoDB} from "@aws-sdk/client-dynamodb"` reduces that overhead to about 3 MB. Restricting the import to just the DynamoDB client and `ListTablesCommand` command reduces the overhead to less than 100 KB.

```
// Load the DynamoDB client and ListTablesCommand command for Node.js
import {
  DynamoDBClient,
  ListTablesCommand
} from "aws-sdk/client-dynamodb";
const dbclient = new DynamoDBClient({});
```

Calling commands in V3

You can perform operations in V3 using either V2 or V3 commands. To use V3 commands you import the commands and the required AWS Services package clients, and run the command using the `.send` method using the async/await pattern.

To use V2 commands you import the required AWS Services packages, and run the V2 command directly in the package using either a callback or async/await pattern.

Using V3 commands

V3 provides a set of commands for each AWS Service package to enable you to perform operations for that AWS Service. After you install an AWS Service, you can browse the available commands in your project's `node-modules/@aws-sdk/client-PACKAGE_NAME/commands folder.`

You must import the commands you want to use. For example, the following code loads the DynamoDB service, and the `CreateTableCommand` command.

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

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

```
  CLIENT.send(newXXXCommand)
```

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

```
import {DynamoDB, CreateTableCommand} from "@aws-sdk/client-dynamodb";
const dynamodb = new DynamoDB({region: 'us-west-2'});
var tableParams = {
    Table : 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.

```
const {DynamoDB} = require('@aws-sdk/client-dynamodb');
```

To call V2 commands in the recommended async/await pattern, use the following syntax.

```
client.command(parameters)
```

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

```
const {DynamoDB} = require('@aws-sdk/client-dynamodb');
const dymamoDB = new DynamoDB({region: 'us-west-2'});
var tableParams = {
    TableName : TABLE_NAME
};
async function run() => {
      try {
           const data = await dymamoDB.createTable(tableParams);
           console.log("Success", data);
      }
      catch (err) {
           console.log("Error", err);
      }
};
run();
```

The following example uses the V2 `createBucket` command to create an Amazon S3 bucket using the callback pattern.

```
const {S3} = require('@aws-sdk/client-s3');
const s3 = new S3({region: 'us-west-2'});
var bucketParams = {
    Bucket : BUCKET_NAME
};
function run(){
         s3.createBucket(bucketParams, function(err, data) {
         if (err) {
```

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

New middleware stack

V2 of the SDK enabled you to modify a request throughout the multiple stages of its lifecycle by attaching event listeners to the request. This approach can make it difficult to debug what went wrong during a request's lifecycle.

In V3, you can use a new middleware stack to control the lifecycle of an operation call. This approach provides a couple of benefits. Each middleware stage in the stack calls the next middleware stage after making any changes to the request object. This also makes debugging issues in the stack much easier, because you can see exactly which middleware stages were called leading up to the error.

The following example adds a custom header to a Amazon DynamoDB client (which we created and showed earlier) using middleware. The first argument is a function that accepts `next`, which is the next middleware stage in the stack to call, and `context`, which is an object that contains some information about the operation being called. The function returns a function that accepts `args`, which is an object that contains the parameters passed to the operation and the request. It returns the result from calling the next middleware with `args`.

```
dbclient.middlewareStack.add(
  (next, context) => args => {
    args.request.headers["Custom-Header"] = "value";
    return next(args);
  },
  {
    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 (p. 53).

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 (p. 23).

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

Using browsers in V3

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

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

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

> **Note**
> For more information about Webpack, see Bundling applications with webpack (p. 42).

To use V2 of the SDK for JavaScript you instead add a script tag that points to the latest version of the V2 SDK. For more information, see https://docs.aws.amazon.com/sdk-for-javascript/v2/developer-guide/getting-started-browser.html#getting-started-browser-run-samplethe SDK for JavaScript v2 Developer Guide.

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
- JavaScript Developer Blog
- AWS JavaScript Forum
- JavaScript examples in the AWS Code Catalog
- AWS Code Example Repository
- Gitter channel
- Stack Overflow
- Stack Overflow questions taggedAWS -sdk-js
- GitHub
  - SDK Source
  - Documentation Source

Getting started with the AWS SDK for JavaScript

The AWS SDK for JavaScript provides access to web services in either browser scripts or Node.js. This section has two 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 Node.js 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 (p. 53).

Topics

Getting started in a browser script

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 (p. 6).

For other examples of using (V3) of the SDK for JavaScript with the Node.js in the browser, see:

- Viewing photos in an Amazon S3 bucket from a browser (p. 174)
- Uploading photos to Amazon S3 from a browser (p. 183)
- Build an app to submit data to DynamoDB (p. 298)

This browser script example shows you:

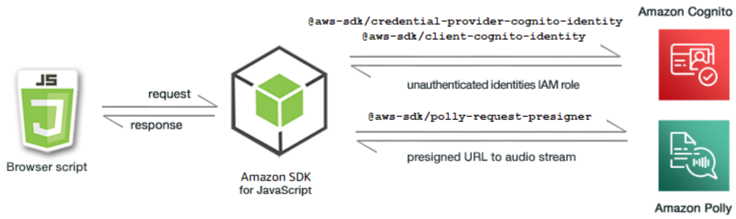
- How to access AWS services from a browser script using Amazon Cognito Identity.
- How to turn text into synthesized speech using Amazon Polly.
- How to use a presigner object to create a presigned URL.

The Scenario

Amazon Polly is a cloud service that converts text into lifelike speech. You can use Amazon Polly to develop applications that increase engagement and accessibility. Amazon Polly supports multiple languages and includes a variety of lifelike voices. For more information about Amazon Polly, see the Amazon Polly Developer Guide.

This example shows you how to set up and run a browser script that takes text, sends that text to Amazon Polly, and returns the URL of the synthesized audio of the text for you to play. The browser script uses an Amazon Cognito Identity pool to provide credentials needed to access AWS services. The example demonstrates the basic patterns for loading and using the SDK for JavaScript in browser scripts.

> **Note**
> You must run this example in a browser that supports HTML 5 audio to playback the synthesized speech.



The browser script uses the SDK for JavaScript to synthesize text by using the following APIs:

- `CognitoIdentityClient` constructor
- `Polly` constructor
- `getSynthesizeSpeechUrl`

Step 1: Create an Amazon Cognito Identity Pool

In this exercise, you create and use an Amazon Cognito Identity pool to provide unauthenticated access to your browser script for the Amazon Polly service. Creating an identity pool also creates two AWS Identity and Access Management (IAM) roles, one to support users authenticated by an identity provider and the other to support unauthenticated guest users.

In this exercise, 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.

To create an Amazon Cognito Identity pool

1. Sign in to the AWS Management Console and open the Amazon Cognito console at Amazon Web Services Console.
2. Choose **Manage Identity Pools** on the console opening page.
3. On the next page, choose **Create new identity pool**.

   > **Note**
   > If there are no other identity pools, the Amazon Cognito console will skip this page and open the next page instead.
4. In the **Getting started wizard**, type a name for your identity pool in **Identity pool name**.
5. Choose **Enable access to unauthenticated identities**.
6. Choose **Create Pool**.
7. On the next page, choose **View Details** to see the names of the two IAM roles created for your identity pool. Make a note of the name of the role for unauthenticated identities. You need this name to add the required policy for Amazon Polly.
8. Choose **Allow**.
9. On the **Sample code** page, select the Platform of *JavaScript*. Then, copy or write down the identity pool ID and the Region. You need these values to replace REGION and IDENTITY_POOL_ID in your browser script.

After you create your Amazon Cognito identity pool, you're ready to add permissions for Amazon Polly that are needed by your browser script.

Step 2: Add a Policy to the Created IAM Role

To enable browser script access to Amazon Polly for speech synthesis, use the unauthenticated IAM role created for your Amazon Cognito identity pool. This requires you to add an IAM policy to the role. For more information about IAM roles, see Creating a Role to Delegate Permissions to an AWS Service in the *IAM User Guide*.

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

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. In the navigation panel on the left of the page, choose **Roles**.
3. In the list of IAM roles, click the link for the unauthenticated identities role previously created by Amazon Cognito.
4. In the **Summary** page for this role, choose **Attach policies**.
5. In the **Attach Permissions** page for this role, find and then select the check box for **AmazonPollyFullAccess**.

   Note
   You can use this process to enable access to any AWS service.
6. Choose **Attach policy**.

After you create your Amazon Cognito identity pool and add permissions for Amazon Polly to your IAM role for unauthenticated users, you are ready to build the webpage and browser script.

Step 3: Create a project environment

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.

Step 4: Create the HTML Page

The sample app consists of a single HTML page that contains the user interface, and a JavaScript file that contains the required JavaScript. To begin, create an HTML document and copy the following contents into it. The page includes an input field and button, an `<audio>` element to play the synthesized speech, and a `<p>` element to display messages. (Note that the full example is shown at the bottom of this page.)

The `<script>` element adds the `main.js` file, which contains all the required JavaScript for the example.

You use webpack to create the `main.js` file, as described in Step 5: Write the JavaScript (p. 11).

For more information about the `<audio>` element, see audio.

The full HTML page is available here on GitHub.

Save the HTML file, naming it `polly.html`. After you have created the user interface for the application, you're ready to add the browser script code that runs the application.

To use V3 of the AWS SDK for JavaScript in the browser, you require Webpack to bundle the JavaScript modules and functions, which you installed in the Step 3: Create a project environment (p. 10).

Note

For information on installing Webpack, see https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/webpack.html.

Step 5: Write the JavaScript

Create a file named `polly.js`, and paste the code below into it. The full JavaScript page is available here on GitHub. The code first imports the required AWS SDK clients and commands. Then it creates the `Polly` service client object, specifying the credentials for the SDK. To synthesize speech with Amazon Polly, it provides a variety of parameters including the sound format of the output, the sampling rate, the ID of the voice to use, and the text to play back. When you initially create the parameters, set the `Text:` parameter to an empty string; the `Text:` parameter will be set to the value you retrieve from the `<input>` element in the webpage.

Next, it creates a function named `speakText()` that is be invoked as an event handler by the button. Amazon Polly returns synthesized speech as an audio stream. The easiest way to play that audio in a browser is to have Amazon Polly make the audio available at a presigned URL you can then set as the `src` attribute of the `<audio>` element in the webpage.

Next it create thes `Presigner` object you'll use to create the presigned URL from which the synthesized speech audio can be retrieved. You must pass the speech parameters that you defined as well as the `Polly` service object that you created to the `Polly.Presigner` constructor.

After it creates the presigner object, it calls the `getSynthesizeSpeechUrl` method of that object, passing the speech parameters. If successful, this method returns the URL of the synthesized speech, which the code then assign to the `<audio>` element for playback.

Finally, from your project folder containing `polly.js` run the following at the command prompt to bundle the JavaScript for this example in a file named `main.js`:

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

Note

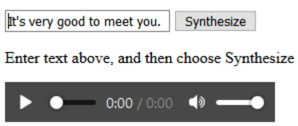
For information about installing webpack, see Bundling applications with webpack (p. 42).

```
import { CognitoIdentityClient } from "@aws-sdk/client-cognito-identity";
import {
    fromCognitoIdentityPool,
} from "@aws-sdk/credential-provider-cognito-identity";
import { Polly } from "@aws-sdk/client-polly";
import { getSynthesizeSpeechUrl } from "@aws-sdk/polly-request-presigner";

// Create the Polly service client, assigning your credentials
const client = new Polly({
    region: "REGION",
    credentials: fromCognitoIdentityPool({
        client: new CognitoIdentityClient({ region: "REGION" }),
        identityPoolId: "IDENTITY_POOL_ID" // IDENTITY_POOL_ID
    }),
});

// Set the parameters
const speechParams = {
    OutputFormat: "OUTPUT_FORMAT", // For example, 'mp3'
    SampleRate: "SAMPLE_RATE", // For example, '16000
    Text: "", // The 'speakText' function supplies this value
    TextType: "TEXT_TYPE", // For example, "text"
    VoiceId: "POLLY_VOICE" // For example, "Matthew"
};
```

```
const speakText = async () => {
    // Update the Text parameter with the text entered by the user
    speechParams.Text = document.getElementById("textEntry").value;
    try{
        let url = await getSynthesizeSpeechUrl({
            client, params: speechParams
        });
        console.log(url);
        // Load the URL of the voice recording into the browser
        document.getElementById('audioSource').src = url;
        document.getElementById('audioPlayback').load();
        document.getElementById('result').innerHTML = "Speech ready to play.";
    } catch (err) {
        console.log("Error", err);
        document.getElementById('result').innerHTML = err;
    }
};
// Expose the function to the browser
window.speakText = speakText;
```

Step 6: Run the Example

To run the example app, load `polly.html` into a web browser. The app should look similar to the following.



Enter a phrase you want turned to speech in the input box, then choose **Synthesize**. When the audio is ready to play, a message appears. Use the audio player controls to hear the synthesized speech.

Possible Enhancements

Here are variations on this application you can use to further explore using the SDK for JavaScript in a browser script.

- Experiment with other sound output formats.
- Add the option to select any of the various voices provided by Amazon Polly.
- Integrate an identity provider like Facebook or Amazon to use with the authenticated IAM role.

Getting started in Node.js

This Node.js code example shows:

- How to install and include the modules that your project uses.
- How to write the Node.js code to create an Amazon S3 bucket and upload an object to that bucket.
- How to run the code.

The scenario

The example shows how to set up and run a simple Node.js module that creates an Amazon S3 bucket, then adds a text object to it.

Prerequisite tasks

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

- Install npm.
- 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.
- You need to provide credentials to AWS so that only your account and its resources are accessed by the SDK. For more information about obtaining your account credentials, see Loading credentials in Node.js from the shared credentials file (p. 33).

Step 1: Install the Amazon S3 package and dependencies

To install the client package and dependencies:

1. In the `src` project directory, there is a `package.json` file for holding the metadata for your Node.js project.

   Note
   For details about using `package.json` in a Node.js project, see What is the file package.json?.

```
{
  "name": "aws-sdk-v3-iam-examples",
  "version": "1.0.0",
  "main": "index.js",
  "dependencies": {
   "@aws-sdk/client-s3": "^3.32.0",
    "@aws-sdk/node-http-handler": "^3.32.0"
  },
  "type": "module"
}
```

   The example code is available here on GitHub.
2. From the `nodegetstarted` directory containing the `package.json` enter the following command.

```
npm install
```

   The packages and dependencies are installed.

   Note
   You can add dependencies to the `package.json` and install them by running `npm install`. You can also add dependencies directly through the command line. For example, to install the AWS SDK for JavaScript v3 client module for Amazon S3, enter the command below in the command line.

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

The `package.json` dependencies are automatically updated.

Step 2: Write the Node.js code

Important
This example 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 (p. 56).

Create a file named `sampleClient.js` to contain the client for creating the Amazon S3 service client object. Copy and paste the code below into it. Replace *`REGION`* with your AWS Region.

```
import { S3Client } from "@aws-sdk/client-s3";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon S3 service client object.
const s3Client = new S3Client({ region: REGION });
export { s3Client };
```

The example code can be found here on GitHub.

First, define the parameters by replacing *`BUCKET_NAME`* with the name of the bucket, *`KEY`* with the name of the new object, *`BODY`* with some content for the new object.

Next, create an Amazon S3 client object. Then create an async wrapper function that runs two try/catch statements in sequence. The first try/catch statement creates the bucket, and the second creates and uploads the new object.

To create the bucket, you create a constant that runs the `CreateBucketCommand` using the `.send` method using the async/await pattern, passing in the name of the new bucket. The await keyword blocks execution of all the code that follows until the bucket is created. If an error occurs, the first `catch` statement returns an error.

To create and upload an object to the new bucket after it is created, you create a constant that runs the `PutObjectCommand`, also using the `.send` method using the async/await pattern, and passing in the bucket, key, and body parameters. If an error occurs, the second `catch` statement returns an error.

```
// Import required AWS SDK clients and commands for Node.js.
import { PutObjectCommand, CreateBucketCommand } from "@aws-sdk/client-s3";
import { s3Client } from "./libs/s3Client.js";

// Set the parameters
const params = {
  Bucket: "BUCKET_NAME", // The name of the bucket. For example, 'sample_bucket_101'.
  Key: "KEY", // The name of the object. For example, 'sample_upload.txt'.
  Body: "BODY", // The content of the object. For example, 'Hello world!".
};

const run = async () => {
  // Create an Amazon S3 bucket.
  try {
    const data = await s3Client.send(
        new CreateBucketCommand({ Bucket: params.Bucket })
    );
    console.log(data);
```

```
    console.log("Successfully created a bucket called ", data.Location);
    return data; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
  // Create an object and upload it to the Amazon S3 bucket.
  try {
    const results = await s3Client.send(new PutObjectCommand(params));
    console.log(
        "Successfully created " +
        params.Key +
        " and uploaded it to " +
        params.Bucket +
        "/" +
        params.Key
    );
    return results; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

The example code can be found here on GitHub.

Step 3: Run the example

Enter the following command to run the example.

```
node sample.js
```

If the upload is successful, you'll see a confirmation message at the command prompt. You can also find the bucket and the uploaded text object in the Amazon S3 console.

Getting started in React Native

This tutorial shows you how you can create a React Native app using React Native CLI.

This tutorial shows you:

- How to install and include the AWS SDK for JavaScript version 3 (V3) modules that your project uses.
- How to write code that connects to Amazon Simple Storage Service (Amazon S3) to create and delete an Amazon S3 bucket.

The Scenario

Amazon S3 is a cloud service that enables you to store and retrieve any amount of data at any time, from anywhere on the web. React Native is a development framework that enables you to create mobile applications. This tutorial shows you how you can create a React Native app that connects to Amazon S3 to create and delete an Amazon S3 bucket.

The app uses the following SDK for JavaScript APIs:

- `CognitoIdentityClient` constructor
- `S3` constructor

Setup for this tutorial

This section provides the minimal setup needed to complete this tutorial. You shouldn't consider this to be a full setup. For that, see Setting up the SDK for JavaScript (p. 23).

> **Note**
> If you've already completed any of the following steps through other tutorials or existing configuration, skip those steps.

Create an AWS account

To create an AWS account, see How do I create and activate a new Amazon Web Services account?

Create AWS credentials and a profile

To perform these tutorials, you need to create an AWS Identity and Access Management (IAM) user and obtain credentials for that user. After you have those credentials, you make them available to the SDK in your development environment. Here's how.

To create and use credentials

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. Choose **Users**, and then choose **Add user**.
3. Provide a user name. For this tutorial, we'll use *React-Native-Tutorial-User*.
4. Under **Select AWS access type**, select **Programmatic access**, and then choose **Next: Permissions**.
5. Choose **Attach existing policies directly**.
6. In **Search**, enter **`s3`**, and then select **AmazonS3FullAccess**.
7. Choose **Next: Tags**, **Next: Review**, and **Create user**.
8. Record the credentials for *React-Native-Tutorial-User*. You can do so by downloading the `.csv` file or by copying and pasting the *Access key ID* and *Secret access key*.

    > **Warning**
    > Use appropriate security measures to keep these credentials safe and rotated.
9. Create or open the shared AWS credentials file. This file is `~/.aws/credentials` on Linux and macOS systems, and `%USERPROFILE%\.aws\credentials` on Windows.
10. Add the following text to the shared AWS credentials file, but replace the example ID and example key with the ones you obtained earlier. Remember to save the file.

```
[javascript-tutorials]
aws_access_key_id = AKIAIOSFODNN7EXAMPLE
aws_secret_access_key = wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
```

The preceding procedure is the simplest of several possibilities for authentication and authorization. For complete information, see Setting credentials (p. 31).

Install other tools

To complete this tutorial, you need to set up your React Native development environment.

You also need to install the following tools:

- `Node.js`
- Xcode if you're testing on IOS.
- Android Studio if you're testing on Android.

Step 1: Create an Amazon Cognito Identity Pool

In this exercise, you create and use an Amazon Cognito Identity pool to provide unauthenticated access to your app for the Amazon S3 service. Creating an identity pool also creates two AWS Identity and Access Management (IAM) roles, one to support users authenticated by an identity provider and the other to support unauthenticated guest users.

In this exercise, 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.

To create an Amazon Cognito Identity pool

1. Sign in to the AWS Management Console and open the Amazon Cognito console at Amazon Web Services Console.
2. Choose **Manage Identity Pools** on the console opening page.
3. On the next page, choose **Create new identity pool**.

   Note
   If there are no other identity pools, the Amazon Cognito console will skip this page and open the next page instead.
4. In the **Getting started wizard**, type a name for your identity pool in **Identity pool name**.
5. Choose **Enable access to unauthenticated identities**.
6. Choose **Create Pool**.
7. On the next page, choose **View Details** to see the names of the two IAM roles created for your identity pool. Make a note of the name of the role for unauthenticated identities. You need this name to add the required policy for Amazon S3.
8. Choose **Allow**.
9. On the **Sample code** page, select the Platform of *JavaScript*. Then, copy or write down the identity pool ID and the Region. You need these values to replace *`REGION`* and *`IDENTITY_POOL_ID`* in your browser script.

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

Step 2: Add a Policy to the Created IAM Role

To enable browser script access to Amazon S3 to create and delete an Amazon S3 bucket, use the unauthenticated IAM role created for your Amazon Cognito identity pool. This requires you to add an IAM policy to the role. For more information about IAM roles, see Creating a Role to Delegate Permissions to an AWS Service in the *IAM User Guide*.

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

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. In the navigation panel on the left of the page, choose **Roles**.

3. In the list of IAM roles, click the link for the unauthenticated identities role previously created by Amazon Cognito.
4. In the **Summary** page for this role, choose **Attach policies**.
5. In the **Attach Permissions** page for this role, find and then select the check box for **AmazonS3FullAccess**.

   Note
   You can use this process to enable access to any AWS service.
6. Choose **Attach policy**.

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 build the app.

Step 3: Create app using create-react-native-app

Create a React Native App by running the following command.

```
npx react-native init ReactNativeApp --npm
```

Step 4: Install the Amazon S3 package and other dependencies

Inside the directory of the project, run the following commands to install the Amazon S3 package.

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

This command installs the Amazon S3 package in your project, and updates `package.json` to list Amazon S3 as a project dependency. You can find information about this package by searching for "@aws-sdk" on the https://www.npmjs.com/npm website.

These packages and their associated code are installed in the `node_modules` subdirectory of your project.

For more information about installing Node.js packages, see Downloading and installing packages locally and Creating Node.js modules on the npm (Node.js package manager) website. For information about downloading and installing the AWS SDK for JavaScript, see Installing the SDK for JavaScript (p. 24).

Install other dependencies required for authentication.

```
npm install @aws-sdk/client-cognito-identity @aws-sdk/credential-provider-cognito-identity
```

Step 5: Write the React Native code

Add the following code to the `App.js`.

```
import React, { useState } from "javascriptv3/example_code/reactnative/App";
 import { Button, StyleSheet, Text, TextInput, View } from "react-native";

 import {
   S3Client,
   CreateBucketCommand,
```

```
  DeleteBucketCommand,
} from "@aws-sdk/client-s3";
import { CognitoIdentityClient } from "@aws-sdk/client-cognito-identity";
import { fromCognitoIdentityPool } from "@aws-sdk/credential-provider-cognito-identity";

const App = () => {
  const [bucketName, setBucketName] = useState("");
  const [successMsg, setSuccessMsg] = useState("");
  const [errorMsg, setErrorMsg] = useState("");

  // Replace REGION with the appropriate AWS Region, such as 'us-east-1'.
  const region = "REGION";
  const client = new S3Client({
    region,
    credentials: fromCognitoIdentityPool({
      client: new CognitoIdentityClient({ region }),
      // Replace IDENTITY_POOL_ID with an appropriate Amazon Cognito Identity Pool ID for,
such as 'us-east-1:xxxxxx-xxx-4103-9936-b52exxxxfd6'.
      identityPoolId: "IDENTITY_POOL_ID",
    }),
  });

  const createBucket = async () => {
    setSuccessMsg("");
    setErrorMsg("");

    try {
      await client.send(new CreateBucketCommand({ Bucket: bucketName }));
      setSuccessMsg(`Bucket "${bucketName}" created.`);
    } catch (e) {
      setErrorMsg(e);
    }
  };

  const deleteBucket = async () => {
    setSuccessMsg("");
    setErrorMsg("");

    try {
      await client.send(new DeleteBucketCommand({ Bucket: bucketName }));
      setSuccessMsg(`Bucket "${bucketName}" deleted.`);
    } catch (e) {
      setErrorMsg(e);
    }
  };

  return (
    <View style={styles.container}>
      <Text style={{ color: "green" }}>
        {successMsg ? `Success: ${successMsg}` : ``}
      </Text>
      <Text style={{ color: "red" }}>
        {errorMsg ? `Error: ${errorMsg}` : ``}
      </Text>
      <View>
        <TextInput
          style={styles.textInput}
          onChangeText={(text) => setBucketName(text)}
          autoCapitalize={"none"}
          value={bucketName}
          placeholder={"Enter Bucket Name"}
        />
        <Button
          backroundColor="#68a0cf"
          title="Create Bucket"
          onPress={createBucket}
```

```
        />
        <Button
          backroundColor="#68a0cf"
          title="Delete Bucket"
          onPress={deleteBucket}
        />
      </View>
    </View>
  );
};

const styles = StyleSheet.create({
  container: {
    flex: 1,
    alignItems: "center",
    justifyContent: "center",
  },
});

export default App;
```

The code first imports required React, React Native, and AWS SDK dependencies.

Inside the function App:

- The S3Client object is created, specifying the credentials using Amazon Cognito Identity Pool created earlier.
- The methods `createBucket` and `deleteBucket` create and delete the specified bucket, respectively.
- The React Native View displays a text input field for the user to specify an Amazon S3 bucket name, and buttons to create and delete the specified Amazon S3 bucket.

The full JavaScript page is available here on GitHub.

Step 6: Run the Example

To run the example, either run web, ios or android command using npm.

Here is an example output of running `ios` command on macOS.

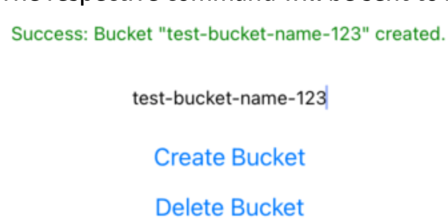
```
$ npm run ios

> ReactNativeApp@0.0.1 ios /Users/trivikr/workspace/ReactNativeApp
> react-native run-ios

info Found Xcode workspace "ReactNativeApp.xcworkspace"
info Launching iPhone 11 (iOS 14.2)
info Building (using "xcodebuild -workspace ReactNativeApp.xcworkspace -configuration Debug
 -scheme ReactNativeApp -destination id=706C1A97-FA38-407D-AD77-CB4FCA9134E9")
success Successfully built the app
info Installing "/Users/trivikr/Library/Developer/Xcode/DerivedData/ReactNativeApp-
cfhmsyhptwflqqejyspdqgjestra/Build/Products/Debug-iphonesimulator/ReactNativeApp.app"
info Launching "org.reactjs.native.example.ReactNativeApp"

success Successfully launched the app on the simulator
```

Here is an example output of running `android` command on macOS.

```
$ npm run android
```

```
> ReactNativeApp@0.0.1 android
> react-native run-android

info Running jetifier to migrate libraries to AndroidX. You can disable it using "--no-jetifier" flag.
Jetifier found 970 file(s) to forward-jetify. Using 12 workers...
info Starting JS server...
info Launching emulator...
info Successfully launched emulator.
info Installing the app...

> Task :app:stripDebugDebugSymbols UP-TO-DATE
Compatible side by side NDK version was not found.

> Task :app:installDebug
02:18:38 V/ddms: execute: running am get-config
02:18:38 V/ddms: execute 'am get-config' on 'emulator-5554' : EOF hit. Read: -1
02:18:38 V/ddms: execute: returning
Installing APK 'app-debug.apk' on 'Pixel_3a_API_30_x86(AVD) - 11' for app:debug
02:18:38 D/app-debug.apk: Uploading app-debug.apk onto device 'emulator-5554'
02:18:38 D/Device: Uploading file onto device 'emulator-5554'
02:18:38 D/ddms: Reading file permision of /Users/trivikr/workspace/ReactNativeApp/android/app/build/outputs/apk/debug/app-debug.apk as: rw-r--r--
02:18:40 V/ddms: execute: running pm install -r -t "/data/local/tmp/app-debug.apk"
02:18:41 V/ddms: execute 'pm install -r -t "/data/local/tmp/app-debug.apk"' on
 'emulator-5554' : EOF hit. Read: -1
02:18:41 V/ddms: execute: returning
02:18:41 V/ddms: execute: running rm "/data/local/tmp/app-debug.apk"
02:18:41 V/ddms: execute 'rm "/data/local/tmp/app-debug.apk"' on 'emulator-5554' : EOF hit.
 Read: -1
02:18:41 V/ddms: execute: returning
Installed on 1 device.

Deprecated Gradle features were used in this build, making it incompatible with Gradle 7.0.
Use '--warning-mode all' to show the individual deprecation warnings.
See https://docs.gradle.org/6.2/userguide/command_line_interface.html#sec:command_line_warnings

BUILD SUCCESSFUL in 6s
27 actionable tasks: 2 executed, 25 up-to-date
info Connecting to the development server...
8081
info Starting the app on "emulator-5554"...
Starting: Intent { cmp=com.reactnativeapp/.MainActivity }
```

Enter the bucket name you want to create or delete and click on either **Create Bucket** or **Delete Bucket**. The respective command will be sent to Amazon S3, and success or error message will be displayed.



Possible Enhancements

Here are variations on this application you can use to further explore using the SDK for JavaScript in a React Native app.

- Add a button to list Amazon S3 buckets, and provide a delete button next to each bucket listed.

- Add a button to put text object into a bucket.
- Integrate an external identity provider like Facebook or Amazon to use with the authenticated IAM role.

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

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

Topics

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 (p. 297).
- 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 SDK for JavaScript supports all modern web browsers, including these minimum versions.

Browser	Version
Google Chrome	49.0+
Mozilla Firefox	45.0+
Opera	36.0+
Microsoft Edge	12.0+
Windows Internet Explorer	N/A
Apple Safari	9.0+
Android Browser	76.0+
UC Browser	12.12+
Samsung Internet	5.0+

Note

Frameworks such as AWS Amplify might not offer the same browser support as the SDK for JavaScript. See the AWS Amplify Documentation for details.

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, enter the following command at the command prompt, where *SERVICE* is the name of a service, such as `s3`.

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

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.

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

Migrating your code to SDK for JavaScript V3

There are a number of migration paths to the SDK for JavaScript version 3 (V3). To take full advantage of the reduction in capacity potential of V3, we recommend using path 3.

Important

AWS SDK for JavaScript version 3 (v3) also comes with modernized interfaces for client configurations and utilities, which include credentials, Amazon S3 multipart upload, DynamoDB document client, waiters, and so forth). 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.

Path 1

Perform minimal changes:

- Install only the specific AWS Service packages you need.
- Create and use V3 service clients, replacing the use of any global configuration values, such as Region, with configuration values passed in as arguments to the client.

Note
You can set the AWS Region using an environment variable, or a shared configuration `config` file. For more information, see Setting credentials in Node.js.

Path 2

Follow path 1 and remove `.promise`, which are not required in V3.

Path 3

Follow path 1 and use the async/await programming model.

Important
For information about significant changes from AWS SDK for JavaScript v2 to v3, please see Upgrading Notes (2.x to 3.x) on GitHub.

The following sections describe these paths in detail, with examples.

Path 1 example

The following code installs the AWS Service package for Amazon S3.

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

The following code loads the Amazon S3 service.

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

Note
To use this approach you must import the full AWS Service packages, `S3` in this case, and not just the service clients.

The following code creates an Amazon S3 service object in the `us-west-2` Region.

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

The following code creates and Amazon S3 bucket using a callback function, using the following syntax from V2.

```
client.command(parameters)
```

```
import {S3} from '@aws-sdk/client-s3';
const client = new S3({region: 'us-west-2'});
const bucketParams = {
    Bucket : BUCKET_NAME
};
function run(){
```

```
        client.createBucket(bucketParams, function(err, data) {
        if (err) {
        console.log("Error", err);
        } else {
        console.log("Success", data.Location);
        }
    })
};
run();
```

Path 2 example

Here is a function call in V2 using a `promise`.

```
const data =  await v2client.command(params).promise()
```

Here is the V3 version.

```
const data = await v3client.command(params)
```

Path 3 examples

The following command installs the AWS Service package for Amazon S3.

```
npm install @aws-sdk/client-s3;
```

The following code loads only the Amazon S3 client, reducing the overhead.

```
import {S3Client, CreateBucketCommand} from '@aws-sdk/client-s3';
```

If you install only the client of a package, you must also import the V3 commands you want to use. In this case, the code imports the `CreateBucketCommand`, which enables you to create an Amazon S3 bucket. You can browse the available commands in your project's `node-modules/@aws-sdk/client-PACKAGE_NAME/commands` folder.

The following code creates an Amazon S3 service client object in the *`us-west-2`* Region.

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

To call imported commands using the recommended async/await pattern, you must import the commands you want to use, and use the following syntax to run the command.

```
  CLIENT.send(newXXXCommand)
```

The following example creates an Amazon S3 bucket using the async/await pattern, using only the client of the Amazon S3 service package to reduce overhead.

```

import {S3Client, CreateBucketCommand} from '@aws-sdk/client-s3';
const client = new S3Client({region: 'us-west-2'});
const bucketParams = {
    Bucket : BUCKET_NAME
};
```

```
(async function () {
      try{
        const data = await client.send(new CreateBucketCommand(bucketParams));
        console.log("Success", data);
      } catch (err) {
        console.log("Error", err);
      }
})();
```

For more examples, see SDK for JavaScript code examples (p. 56).

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 these settings:

- The AWS Region in which you will request services
- The *credentials* that authorize your access to SDK resources

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 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

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.

```
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.

```
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 Loading credentials in Node.js from the shared credentials file (p. 33) or Configuration and credential files in the *AWS Command Line Interface User 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.

Getting your credentials

When you create an AWS account, your account is provided with root credential, or an access key, which consists of the following:

- An access key ID
- A secret access key

For more information about your access keys, see Understanding and getting your security credentials in the *AWS General Reference*.

Access keys consist of an access key ID and secret access key, which are used to sign programmatic requests that you make to AWS. If you don't have access keys, you can create them from the AWS Management Console. As a best practice, do not use the AWS account root user access keys for any task where it's not required. Instead, create a new administrator IAM user with access keys for yourself.

The only time that you can view or download the secret access key is when you create the keys. You cannot recover them later. However, you can create new access keys at any time. You must also have permissions to perform the required IAM actions. For more information, see Permissions required to access IAM resources in the *IAM User Guide*.

To create access keys for an IAM user

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. In the navigation pane, choose **Users**.
3. Choose the name of the user whose access keys you want to create, and then choose the **Security credentials** tab.
4. In the **Access keys** section, choose **Create access key**.
5. To view the new access key pair, choose **Show**. You will not have access to the secret access key again after this dialog box closes. Your credentials will look something like this:
   - Access key ID: AKIAIOSFODNN7EXAMPLE
   - Secret access key: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
6. To download the key pair, choose **Download .csv file**. Store the keys in a secure location. You will not have access to the secret access key again after this dialog box closes.

   Keep the keys confidential in order to protect your AWS account and never email them. Do not share them outside your organization, even if an inquiry appears to come from AWS or Amazon.com. No one who legitimately represents Amazon will ever ask you for your secret key.
7. After you download the `.csv` file, choose **Close**. When you create an access key, the key pair is active by default, and you can use the pair right away.

Related topics

- What is IAM? in the *IAM User Guide*
- AWS security credentials in *AWS General Reference*

Setting credentials

AWS uses credentials to identify who is calling services and whether access to the requested resources is allowed. In AWS, these credentials are typically the access key ID and the secret access key that were created along with your account.

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*.

> **Warning**
> While it is possible to do so, we recommend you not hard code credentials inside an application or browser script. Hard coding credentials poses a risk of exposing your access key ID and secret access key.

For more information about how to manage your access keys, see Best practices for managing AWS access keys in the AWS General Reference.

Topics

Setting credentials in Node.js

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.

You can supply your credentials in order of recommendation:

1. Loaded from AWS Identity and Access Management (IAM) roles for Amazon EC2
2. Loaded from the shared credentials file (`~/.aws/credentials`)
3. Loaded from environment variables
4. Loaded from a JSON file on disk
5. Other credential-provider classes provided by the JavaScript SDK

V3 provides a default credential provider in Node.js. So you are not required to supply a credential provider explicitly. The default credential provider 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. If the resolved credential is from a dynamic source, which means the credential can expire, the SDK will only use the specific source to refresh the credential.

Here's the order of the sources where the default credential provider resolve credentials from:

1. Environment variables
2. The shared credentials file
3. Credentials loaded from the Amazon ECS credentials provider (if applicable)
4. Credentials loaded from AWS Identity and Access Management using the credentials provider of the Amazon EC2 instance (if configured in the instance metadata)

> **Warning**
> We don't recommend hard-coding your AWS credentials in your application. Hard-coding credentials poses a risk of exposing your access key ID and secret access key.

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

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*.

Loading credentials in Node.js from the shared credentials file

You can keep your AWS credentials data in a shared file used by SDKs and the command line interface. When the SDK for JavaScript loads, it automatically searches the shared credentials file, which is named "credentials". Where you keep the shared credentials file depends on your operating system:

- The shared credentials file on Linux, Unix, and macOS: `~/.aws/credentials`
- The shared credentials file on Windows: `C:\Users\USER_NAME\.aws\credentials`

If you do not already have a shared credentials file, see Getting your credentials (p. 30). Once you follow those instructions, you should see text similar to the following in the credentials file, where *<YOUR_ACCESS_KEY_ID>* is your access key ID and *<YOUR_SECRET_ACCESS_KEY>* is your secret access key. Create a shared credentials file like below in the directory.

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

The `[default]` section heading specifies a default profile and associated values for credentials. You can create additional profiles in the same shared configuration file, each with its own credential information. The following example shows a configuration file with the default profile and two additional profiles:

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

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

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

By default, the SDK checks the `AWS_PROFILE` environment variable to determine which profile to use. If the `AWS_PROFILE` variable is not set in your environment, the SDK uses the credentials for the `[default]` profile. To use one of the alternate profiles, set or change the value of the `AWS_PROFILE` environment variable. For example, given the configuration file shown, to use the credentials from the work account, set the `AWS_PROFILE` environment variable to `work-account` (as appropriate for your operating system).

Note

When setting environment variables, be sure to take appropriate actions afterward (according to the needs of your operating system) to make the variables available in the shell or command environment.

After setting the environment variable (if needed), you can run a file named `script.js` that uses the SDK as follows.

```
$ node script.js
```

You can also explicitly select the profile used by a client, as shown in the following example.

```
const {fromIni} = require("@aws-sdk/credential-provider-ini");
```

```
const s3Client = new S3.S3Client({
  credentials: fromIni({profile: 'work-account'})
});
```

Loading credentials in Node.js from environment variables

The SDK automatically detects AWS credentials set as variables in your environment and uses them for SDK requests. This eliminates the need to manage credentials in your application. The environment variables that you set to provide your credentials are:

- `AWS_ACCESS_KEY_ID`
- `AWS_SECRET_ACCESS_KEY`
- `AWS_SESSION_TOKEN` (Optional)

> **Note**
> When setting environment variables, be sure to take appropriate actions afterward (according to the needs of your operating system) to make the variables available in the shell or command environment.

Loading credentials in Node.js using a configured credential process

For details about specifying a credential process in the shared AWS `config` file or the shared credentials file, see Sourcing credentials from external processes.

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
3. Hard coding in the script

> **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

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 https://docs.aws.amazon.com/cognito/latest/developerguide/tutorial-create-identity-pool.html.

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 cendentials from the identity pool. This is shown in the following example of creating an Amazon S3 client in the `us-west-2` AWS Region for users in the *`IDENTITY_POOL_ID`* identity pool.

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

const REGION = AWS_REGION

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

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

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

```
    logins: {
      loginData
    }
  })
});

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

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:

```
// 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.

```
// 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.

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

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

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

For a complete list of all built-in modules that Node.js provides, see Node.js v6.11.1 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.

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

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.

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

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { NodeHttpHandler } from "@aws-sdk/node-http-handler";
import ProxyAgent from "proxy-agent";

const proxyAgent = new ProxyAgent("http://internal.proxy.com");
const dynamodbClient = new DynamoDBClient({
    requestHandler: new NodeHttpHandler({
```

```
        httpAgent: proxyAgent,
        httpsAgent: proxyAgent
    }),
});
```

Note

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

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.

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

const certs = [
  fs.readFileSync("/path/to/cert.pem")
];

const dynamodbClient = new DynamoDBClient({
  httpOptions: {
    agent: new https.Agent({
      rejectUnauthorized: true,
      ca: certs
    })
  }
});
```

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 (p. 42). For an example that uses Webpack to load V3 SDK for JavaScript into a browser, see Build an app to submit data to DynamoDB (p. 298).

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 mimimize 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 (p. 42).

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.

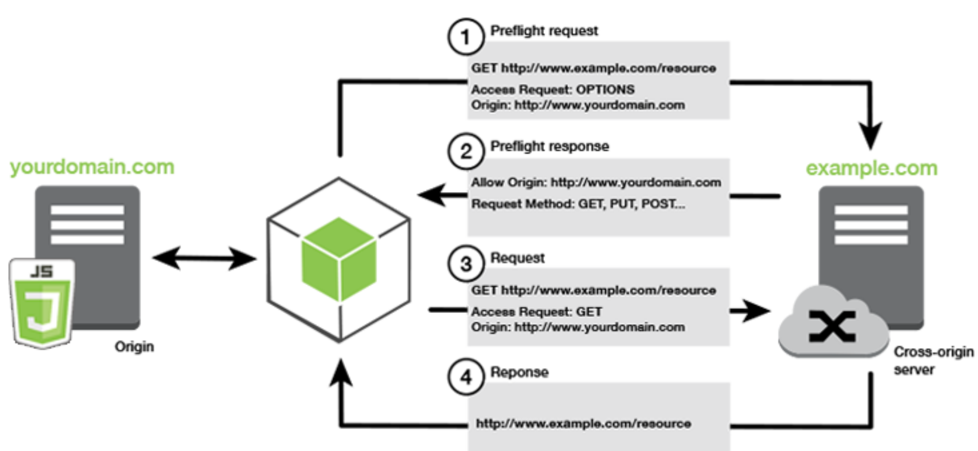
CORS determines whether to allow sharing of resources in a cross-origin request based on the following:

- 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 Conole, the CORS configuration must be JSON.

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

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

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

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

CORS configuration example

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

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

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<CORSConfiguration xmlns="http://s3.amazonaws.com/doc/2006-03-01/">
  <CORSRule>
    <AllowedOrigin>https://example.org</AllowedOrigin>
    <AllowedMethod>HEAD</AllowedMethod>
    <AllowedMethod>GET</AllowedMethod>
    <AllowedMethod>PUT</AllowedMethod>
    <AllowedMethod>POST</AllowedMethod>
```

```
    <AllowedMethod>DELETE</AllowedMethod>
    <AllowedHeader>*</AllowedHeader>
    <ExposeHeader>ETag</ExposeHeader>
    <ExposeHeader>x-amz-meta-custom-header</ExposeHeader>
  </CORSRule>
</CORSConfiguration>
```

JSON

```
[
    {
        "AllowedHeaders": [
            "*"
        ],
        "AllowedMethods": [
            "HEAD",
            "GET",
            "PUT",
            "POST",
            "DELETE"
        ],
        "AllowedOrigins": [
            "https://www.example.org"
        ],
        "ExposeHeaders": [
             "ETag",
             "x-amz-meta-custom-header"]
    }
]
```

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

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

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 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.

> **Note**
> Webpack configuration requirments vary depending on the version of Webpack you install. For more information, see the Webpack documentation.

```
// Import path for resolving file paths
var path = require("path");
module.exports = {
  // Specify the entry point for our app.
  entry: [path.join(__dirname, "browser.js")],
  // Specify the output file containing our bundled code.
  output: {
    path: __dirname,
    filename: 'bundle.js'
  },
   // Enable WebPack to use the 'path' package.
   resolve:{
  fallback: { path: require.resolve("path-browserify")}
  }
  /**
  * In Webpack version v2.0.0 and earlier, you must tell
  * webpack how to use "json-loader" to load 'json' files.
  * To do this Enter 'npm --save-dev install json-loader' at the
  * command line to install the "json-loader' package, and include the
  * following entry in your webpack.config.js.
  * module: {
    rules: [{test: /\.json$/, use: use: "json-loader"}]
  }
  **/
};
```

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

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`.

```
{
  "name": "aws-webpack",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "webpack"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "@aws-sdk/client-iam": "^3.32.0",
    "@aws-sdk/client-s3": "^3.32.0"
  },
  "devDependencies": {
    "webpack": "^5.0.0"
  }
}
```

To build your application, enter the following command.

```
npm run build
```

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

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.

```
<!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.

```
target: "node"
```

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

```
// Import path for resolving file paths
var path = require("path");
module.exports = {
  // Specify the entry point for our app.
  entry: [path.join(__dirname, "browser.js")],
```

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

Working with services in the SDK for JavaScript

The AWS SDK for JavaScript 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

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

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

```
// Import the Amazon S3 service client
import {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 (p. 51).

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

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

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

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

Calling services asychronously

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

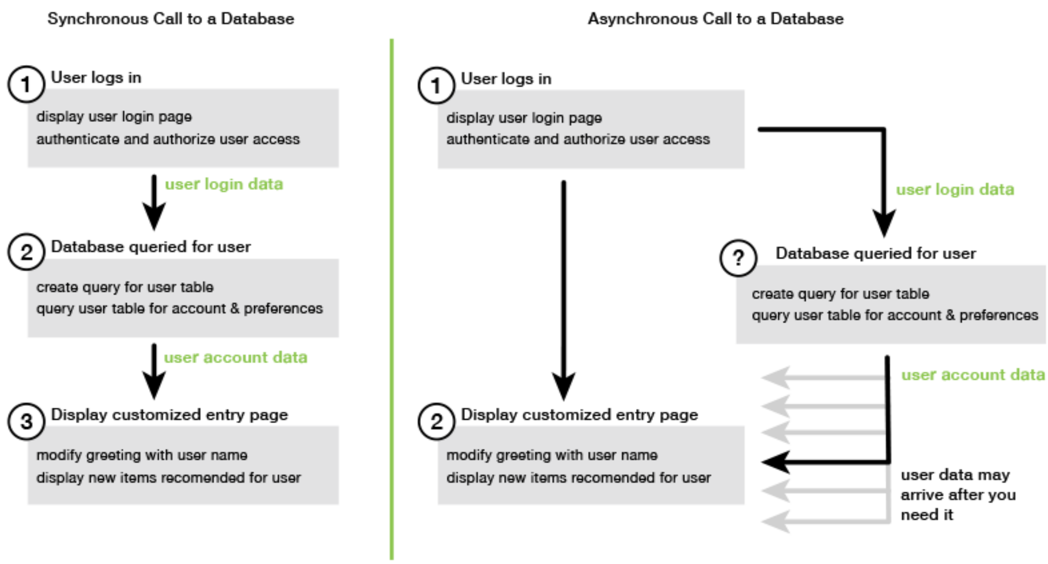
Managing asychronous calls

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 user name and password.
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 asychronously. 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 Loading credentials in Node.js from the shared credentials file (p. 33).

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

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

```
})();
```

Note

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

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`.

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

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

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.

```
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 compatability 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.

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

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

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

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

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

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

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 (p. 4).

To send a request:

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

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

```
  import {
```

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

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

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

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 (p. 50), the `ListTablesCommand` command returned the table names in the `TableNames` property of the response.

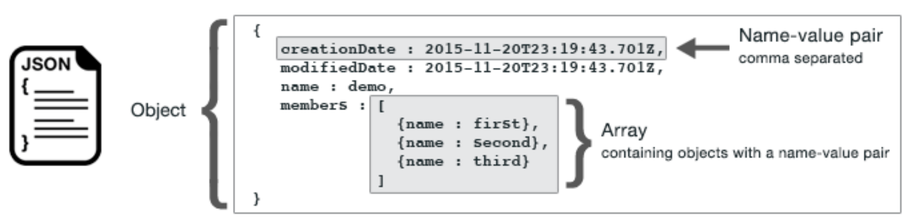
Accessing error information

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.



JSON represents data in two ways:

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

- As an *array*, which is an ordered collection of values. An array is defined within left (`[`) and right (`]`) brackets. Items in the array are comma separated.

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

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

JSON as service object parameters

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

```
const params = {
   FunctionName : "slotPull",
   InvocationType : "RequestResponse",
   LogType : "None"
};
```

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`, `InvocationType`, 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.

```
  import { LambdaClient,
   InvokeCommand } from "@aws-sdk/client-lambda";
  (async function() {
  const lambdaClient = new LambdaClient({ region: "us-west-2" });
  // create JSON object for service call parameters
  const params = {
    FunctionName : "slotPull",
    InvocationType : "RequestResponse",
    LogType : "None"
  };

  // create InvokeCommand command
  const command = new InvokeCommand(params);

  // invoke Lambda function
  try {
    const response = await lambdaClient.send(command);
    console.log(response);
  } catch (err) {
    console.err(err);
  }
})();
```

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

Start to use AWS Cloud9 by signing in to the AWS Cloud9 console as an AWS Identity and Access Management (IAM) entity (for example, an IAM user) who has access permissions for AWS Cloud9 in your AWS account.

To set up an IAM entity in your AWS account to access AWS Cloud9, and to sign in to the AWS Cloud9 console, see Team setup for AWS Cloud9 in the *AWS Cloud9 User Guide*.

Step 2: Set up your AWS Cloud9 development environment

After you sign in to the AWS Cloud9 console, use the console to create an AWS Cloud9 development environment. After you create the environment, AWS Cloud9 opens the IDE for that environment.

See Creating an environment in AWS Cloud9 in the *AWS Cloud9 User Guide* for details.

> **Note**
> As you create your environment in the console for the first time, we recommend that you choose the option to **Create a new instance for environment (EC2)**. This option tells AWS Cloud9 to create an environment, launch an Amazon EC2 instance, and then connect the new instance to the new environment. This is the fastest way to begin using AWS Cloud9.

Step 3: Set up the SDK for JavaScript

After AWS Cloud9 opens the IDE for your development environment, follow one or both of the following procedures to use the IDE to set up the SDK for JavaScript in your environment.

To set up the SDK for JavaScript for Node.js

1. If the terminal isn't already open in the IDE, open it. To do this, on the menu bar in the IDE, choose **Window, New Terminal**.
2. Run the following command to use `npm` to install the `Cloud9` client of the SDK for JavaScript.

```
npm install @aws-sdk/client-cloud9
```

If the IDE can't find `npm`, run the following commands, one at a time in the following order, to install `npm`. (These commands assume you chose the option to **Create a new instance for environment (EC2)**, earlier in this topic.)

Warning
AWS does not control the following code. Before you run it, be sure to verify its authenticity and integrity. More information about this code can be found in the nvm (Node Version Manager) GitHub repository.

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.34.0/install.sh | bash #
 Download and install Node Version Manager (nvm).
. ~/.bashrc                                                                   #
 Activate nvm.
nvm install node                                                              # Use
 nvm to install npm (and Node.js at the same time).
```

To set up the SDK for JavaScript in the browser

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

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

Note
For more information about Webpack, see Bundling applications with webpack (p. 42)

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 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 *Proposing new code examples* section in the Readme 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 (p. 56) for conversion guidelines.

Topics

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 sytax using the following guidelines:

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

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

  To its CommonJS equivelent:

```
const { CreateBucketCommand } = require("@aws-sdk/client-s3");
const { s3 } = require("./libs/s3Client.js");
```

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

```
export {s3}
```

  To its CommonJS equivelent:

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

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

ES6

libs/s3Client.js

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

s3_createbucket.js

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

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

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

// Create the Amazon S3 bucket.
```

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

CommonJS

libs/s3Client.js

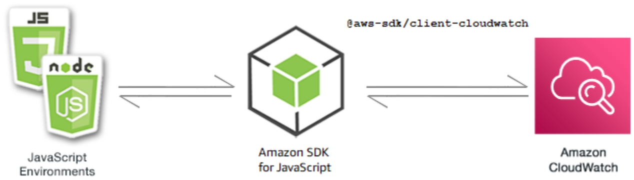
```
// Create service client module using CommonJS syntax.
 const { S3Client } = require("@aws-sdk/client-s3");
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
 // Create Amazon S3 service object.
const s3 = new S3Client({ region: REGION });
// Export 's3' constant.
 module.exports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

s3_createbucket.js

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

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

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

Amazon CloudWatch examples

Amazon CloudWatch (CloudWatch) is a web service that monitors your Amazon Web Services resources and applications you run on AWS in real time. You can use CloudWatch to collect and track metrics, which are variables you can measure for your resources and applications. CloudWatch alarms send notifications or automatically make changes to the resources you are monitoring based on rules that you define.



The JavaScript API for CloudWatch is exposed through the `CloudWatch`, `CloudWatchEvents`, and `CloudWatchLogs` client classes. For more information about using the CloudWatch client classes, see Class: CloudWatch, Class: CloudWatchEvents, and Class: CloudWatchLogs in the *Amazon CloudWatch API reference*.

Topics

Creating alarms in Amazon CloudWatch

This Node.js code example shows:

- How to retrieve basic information about your CloudWatch alarms.
- How to create and delete a CloudWatch alarm.

The scenario

An alarm watches a single metric over a time period you specify, and performs one or more actions based on the value of the metric relative to a given threshold over a number of time periods.

In this example, a series of Node.js modules are used to create alarms in CloudWatch. The Node.js modules use the SDK for JavaScript to create alarms using these methods of the `CloudWatch` client class:

- `DescribeAlarmsCommand`
- `PutMetricAlarmCommand`
- `DeleteAlarmsCommand`

For more information about CloudWatch alarms, see Creating Amazon CloudWatch alarms in the *Amazon CloudWatch User 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 Loading credentials in Node.js from the shared credentials file (p. 33).

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 sytax, please refer to JavaScript ES6/CommonJS syntax (p. 56)

Describing alarms

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

```
import { CloudWatchClient } from "@aws-sdk/client-cloudwatch";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon CloudWatch service client object.
export const cwClient = new CloudWatchClient({ region: REGION });
```

This code is available here on GitHub.

Create a Node.js module with the file name `describeAlarms.js`. Be sure to configure the SDK as previously shown, including downloading the `CloudWatch` client. Create a JSON object to hold the parameters for retrieving alarm descriptions, limiting the alarms returned to those with a state of `INSUFFICIENT_DATA`. Then call the `DescribeAlarmsCommand` method of the `CloudWatch` client service object.

```
// Import required AWS SDK clients and commands for Node.js
import { DescribeAlarmsCommand } from "@aws-sdk/client-cloudwatch";
import { cwClient } from "./libs/cloudWatchClient.js";

// Set the parameters
export const params = { StateValue: "INSUFFICIENT_DATA" };

export const run = async () => {
  try {
    const data = await cwClient.send(new DescribeAlarmsCommand(params));
    console.log("Success", data);
    return data;
    data.MetricAlarms.forEach(function (item, index, array) {
      console.log(item.AlarmName);
      return data;
    });
  } catch (err) {
    console.log("Error", err);
  }
};
// Uncomment this line to run execution within this file.
// run();
```

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

```
node describeAlarms.js
```

This example code can be found here on GitHub.

Creating an alarm for a CloudWatch metric

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

```
import { CloudWatchClient } from "@aws-sdk/client-cloudwatch";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon CloudWatch service client object.
export const cwClient = new CloudWatchClient({ region: REGION });
```

This code is available here on GitHub.

Create a Node.js module with the file name `putMetricAlarm.js`. Be sure to configure the SDK as previously shown, including installing the required clients and packages. Create a JSON object for the parameters needed to create an alarm based on a metric, in this case the CPU utilization of an Amazon EC2 instance. The remaining parameters are set so the alarm triggers when the metric exceeds a threshold of 70 percent. Then call the `DescribeAlarmsCommand` method of the `CloudWatch` client service object.

Note
Replace *INSTANCE_ID* with the ID of the Amazon EC2 instance.

```
// Import required AWS SDK clients and commands for Node.js
import { PutMetricAlarmCommand } from "@aws-sdk/client-cloudwatch";
import { cwClient } from "./libs/cloudWatchClient.js";

// Set the parameters
export const 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",
};

export const run = async () => {
  try {
    const data = await cwClient.send(new PutMetricAlarmCommand(params));
    console.log("Success", data);
    return data;
```

```
  } catch (err) {
    console.log("Error", err);
  }
};
// Uncomment this line to run execution within this file.
// run();
```

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

```
node putMetricAlarm.js
```

This example code can be found here on GitHub.

Deleting an alarm

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

```
import { CloudWatchClient } from "@aws-sdk/client-cloudwatch";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon CloudWatch service client object.
export const cwClient = new CloudWatchClient({ region: REGION });
```

This code is available here on GitHub.

Create a Node.js module with the file name `deleteAlarms.js`. Be sure to configure the SDK as previously shown, including downloading the CloudWatch client. Create a JSON object to hold the names of the alarms to delete. Then call the `DeleteAlarmsCommand` method of the `CloudWatch` client service object.

> **Note**
> Replace *ALARM_NAMES* with the names of the alarms.

```
// Import required AWS SDK clients and commands for Node.js
import { DeleteAlarmsCommand } from "@aws-sdk/client-cloudwatch";
import { cwClient } from "./libs/cloudWatchClient.js";

// Set the parameters
export const params = { AlarmNames: "ALARM_NAME" }; // e.g., "Web_Server_CPU_Utilization"

export const run = async () => {
  try {
    const data = await cwClient.send(new DeleteAlarmsCommand(params));
    console.log("Success, alarm deleted; requestID:", data);
    return data;
  } catch (err) {
    console.log("Error", err);
  }
};

// Uncomment this line to run execution within this file.
// run();
```

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

```
node deleteAlarms.js
```

This example code can be found here on GitHub.

Using alarm actions in Amazon CloudWatch

This Node.js code example shows:

- How to change the state of your Amazon EC2 instances automatically based on a CloudWatch alarm.

The scenario

Using alarm actions, you can create alarms that automatically stop, terminate, reboot, or recover your Amazon EC2 instances. You can use the stop or terminate actions when you no longer need an instance to be running. You can use the reboot and recover actions to automatically reboot those instances.

In this example, a series of Node.js modules are used to define an alarm action in CloudWatch that triggers the reboot of an Amazon EC2 instance. The Node.js modules use the SDK for JavaScript to manage Amazon EC2 instances using these methods of the `CloudWatch` client class:

- `EnableAlarmActionsCommand`
- `DisableAlarmActionsCommand`

For more information about CloudWatch alarm actions, see Create alarms to stop, terminate, reboot, or recover an instance in the *Amazon CloudWatch User 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 Loading credentials in Node.js from the shared credentials file (p. 33).
- Create an IAM role whose policy grants permission to describe, reboot, stop, or terminate an Amazon EC2 instance. For more information about creating an IAM role, see Creating a role to delegate permissions to an AWS service in the *IAM 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 sytax, please refer to JavaScript ES6/CommonJS syntax (p. 56)

Use the following role policy when creating the IAM role.

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

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

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 (p. 56).

Creating and enabling actions on an alarm

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

```
import { CloudWatchClient } from "@aws-sdk/client-cloudwatch";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon CloudWatch service client object.
export const cwClient = new CloudWatchClient({ region: REGION });
```

This code is available here on GitHub.

Create a Node.js module with the file name `enableAlarmActions.js`. Be sure to configure the SDK as previously shown, including downloading the CloudWatch client.

Create a JSON object to hold the parameters for creating an alarm, specifying `ActionsEnabled` as `true` and an array of Amazon Resource Names (ARNs) for the actions the alarm will trigger. Call the `PutMetricAlarmCommand` method of the `CloudWatch` client service object, which creates the alarm if it does not exist or updates it if the alarm does exist.

In the callback function for the `PutMetricAlarmCommand`, on successful completion create a JSON object containing the name of the CloudWatch alarm. Call the `EnableAlarmActionsCommand` method to enable the alarm action.

Note
Replace *`ALARM_NAME`* with the name of the alarm, and *`INSTANCE_ID`* with the ID of the Amazon EC2 instance.

```
// Import required AWS SDK clients and commands for Node.js
```

```
import {
  PutMetricAlarmCommand,
  EnableAlarmActionsCommand,
} from "@aws-sdk/client-cloudwatch";
import { cwClient } from "./libs/cloudWatchClient.js";

// Set the parameters
export const params = {
  AlarmName: "ALARM_NAME", //ALARM_NAME
  ComparisonOperator: "GreaterThanThreshold",
  EvaluationPeriods: 1,
  MetricName: "CPUUtilization",
  Namespace: "AWS/EC2",
  Period: 60,
  Statistic: "Average",
  Threshold: 70.0,
  ActionsEnabled: true,
  AlarmActions: ["ACTION_ARN"], //e.g., "arn:aws:automate:us-east-1:ec2:stop"
  AlarmDescription: "Alarm when server CPU exceeds 70%",
  Dimensions: [
    {
      Name: "InstanceId",
      Value: "INSTANCE_ID",
    },
  ],
  Unit: "Percent",
};

export const run = async () => {
  try {
    const data = await cwClient.send(new PutMetricAlarmCommand(params));
    console.log("Alarm action added; RequestID:", data);
    return data;
    const paramsEnableAlarmAction = {
      AlarmNames: [params.AlarmName],
    };
    try {
      const data = await cwClient.send(
        new EnableAlarmActionsCommand(paramsEnableAlarmAction)
      );
      console.log("Alarm action enabled; RequestID:", data.$metadata.requestId);
    } catch (err) {
      console.log("Error", err);
      return data;
    }
  } catch (err) {
    console.log("Error", err);
  }
};
// Uncomment this line to run execution within this file.
// run();
```

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

```
node enableAlarmActions.js
```

This example code can be found here on GitHub.

Disabling actions on an alarm

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

```
import { CloudWatchClient } from "@aws-sdk/client-cloudwatch";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon CloudWatch service client object.
export const cwClient = new CloudWatchClient({ region: REGION });
```

This code is available here on GitHub.

Create a Node.js module with the file name `disableAlarmActions.js`. Be sure to configure the SDK as previously shown, including downloading the CloudWatch client. Create a JSON object containing the name of the CloudWatch alarm. Call the `DisableAlarmActionsCommand` method to disable the actions for this alarm.

> **Note**
> Replace *`ALARM_NAME`* with the alarm names.

```
// Import required AWS SDK clients and commands for Node.js
import { DisableAlarmActionsCommand } from "@aws-sdk/client-cloudwatch";
import { cwClient } from "./libs/cloudWatchClient.js";

// Set the parameters
export const params = { AlarmNames: "ALARM_NAME" }; // e.g., "Web_Server_CPU_Utilization"

export const run = async () => {
  try {
    const data = await cwClient.send(new DisableAlarmActionsCommand(params));
    console.log("Success, alarm disabled:", data);
    return data;
  } catch (err) {
    console.log("Error", err);
  }
};
// Uncomment this line to run execution within this file.
// run();
```

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

```
node disableAlarmActions.js
```

This example code can be found here on GitHub.

Getting metrics from Amazon CloudWatch

This Node.js code example shows:

- How to retrieve a list of published CloudWatch metrics.
- How to publish data points to CloudWatch metrics.

The scenario

Metrics are data about the performance of your systems. You can enable detailed monitoring of some resources, such as your Amazon EC2 instances, or your own application metrics.

In this example, a series of Node.js modules are used to get metrics from CloudWatch and to send events to Amazon CloudWatch Events. The Node.js modules use the SDK for JavaScript to get metrics from CloudWatch using these methods of the `CloudWatch` client class:

- `ListMetricsCommand`
- `PutMetricDataCommand`

For more information about CloudWatch metrics, see Using Amazon CloudWatch metrics in the *Amazon CloudWatch User 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 Loading credentials in Node.js from the shared credentials file (p. 33).

> **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 sytax, please refer to JavaScript ES6/CommonJS syntax (p. 56)

Listing metrics

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

```
import { CloudWatchClient } from "@aws-sdk/client-cloudwatch";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon CloudWatch service client object.
export const cwClient = new CloudWatchClient({ region: REGION });
```

This code is available here on GitHub.

Create a Node.js module with the file name `listMetrics.js`. Be sure to configure the SDK as previously shown, including downloading the CloudWatch client. Create a JSON object containing the parameters needed to list metrics. Call the `ListMetricsCommand` method to list the `IncomingLogEvents` metric.

```
// Import required AWS SDK clients and commands for Node.js
import { ListMetricsCommand } from "@aws-sdk/client-cloudwatch";
import { cwClient } from "./libs/cloudWatchClient.js";
```

```
// Set the parameters
export const params = {
  Dimensions: [
    {
      Name: "LogGroupName" /* required */,
    },
  ],
  MetricName: "IncomingLogEvents",
  Namespace: "AWS/Logs",
};

export const run = async () => {
  try {
    const data = await cwClient.send(new ListMetricsCommand(params));
    console.log("Success. Metrics:", JSON.stringify(data.Metrics));
    return data;
  } catch (err) {
    console.log("Error", err);
  }
};
// Uncomment this line to run execution within this file.
// run();
```

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

```
node listMetrics.js
```

This example code can be found here on GitHub.

Submitting custom metrics

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

```
import { CloudWatchClient } from "@aws-sdk/client-cloudwatch";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon CloudWatch service client object.
export const cwClient = new CloudWatchClient({ region: REGION });
```

This code is available here on GitHub.

Create a Node.js module with the file name `putMetricData.js`. Be sure to configure the SDK as previously shown, including downloading the CloudWatch client. Create a JSON object containing the parameters needed to submit a data point for the `PAGES_VISITED` custom metric. Call the `PutMetricDataCommand` method.

```
// Import required AWS SDK clients and commands for Node.js
import { PutMetricDataCommand } from "@aws-sdk/client-cloudwatch";
import { cwClient } from "./libs/cloudWatchClient.js";

// Set the parameters
export const params = {
  MetricData: [
    {
      MetricName: "PAGES_VISITED",
      Dimensions: [
```

```
        {
          Name: "UNIQUE_PAGES",
          Value: "URLS",
        },
      ],
      Unit: "None",
      Value: 1.0,
    },
  ],
  Namespace: "SITE/TRAFFIC",
};

export const run = async () => {
  try {
    const data = await cwClient.send(new PutMetricDataCommand(params));
    console.log("Success", data.$metadata.requestId);
    return data;
  } catch (err) {
    console.log("Error", err);
  }
};
// Uncomment this line to run execution within this file.
// run();
```

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

```
node putMetricData.js
```

This example code can be found here on GitHub.

Sending events to Amazon CloudWatch Events

This Node.js code example shows:

- How to create and update a rule used to trigger an event.
- How to define one or more targets to respond to an event.
- How to send events that are matched to targets for handling.

The scenario

CloudWatch Events delivers a near real-time stream of system events that describe changes in Amazon Web Services resources to any of various targets. Using simple rules, you can match events and route them to one or more target functions or streams.

In this example, a series of Node.js modules are used to send events to CloudWatch Events. The Node.js modules use the SDK for JavaScript to manage instances using these methods of the `CloudWatchEvents` client class:

- `PutRuleCommand`
- `PutTargetsCommand`
- `PutEventsCommand`

For more information about CloudWatch Events, see Adding events with PutEvents in the *Amazon CloudWatch Events User 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 Loading credentials in Node.js from the shared credentials file (p. 33).
- Create an AWS Lambda function using the **hello-world** blueprint to serve as the target for events. To learn how, see Step 1: Create an Lambda function in the *Amazon CloudWatch Events User Guide*.
- Create an IAM role whose policy grants permission to CloudWatch Events and that includes `events.amazonaws.com` as a trusted entity. For more information about creating an IAM role, see Creating a role to delegate permissions to an AWS service in the *IAM User Guide*.

Use the following role policy when creating the IAM role.

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "CloudWatchEventsFullAccess",
         "Effect": "Allow",
         "Action": "events:*",
         "Resource": "*"
      },
      {
         "Sid": "IAMPassRoleForCloudWatchEvents",
         "Effect": "Allow",
         "Action": "iam:PassRole",
         "Resource": "arn:aws:iam::*:role/AWS_Events_Invoke_Targets"
      }
   ]
}
```

Use the following trust relationship when creating the IAM role.

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

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 (p. 56).

Creating a scheduled rule

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

```
import { CloudWatchEventsClient } from "@aws-sdk/client-cloudwatch-events";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon CloudWatch service client object.
export const cweClient = new CloudWatchEventsClient({ region: REGION });
```

This code is available here on GitHub.

Create a Node.js module with the file name `putRule.js`. Be sure to configure the SDK as previously shown, including downloading the CloudWatch client. To access CloudWatch Events, create an `CloudWatchEvents` client service object. Create a JSON object containing the parameters needed to specify the new scheduled rule, which include the following:

- A name for the rule
- The ARN of the IAM role you created previously
- An expression to schedule triggering of the rule every five minutes

Call the `PutRuleCommand` method to create the rule. The callback returns the ARN of the new or updated rule.

```
// Import required AWS SDK clients and commands for Node.js
import { PutRuleCommand } from "@aws-sdk/client-cloudwatch-events";
import { cweClient } from "./libs/cloudWatchEventsClient.js";

// Set the parameters
export const params = {
  Name: "DEMO_EVENT",
  RoleArn: "IAM_ROLE_ARN", //IAM_ROLE_ARN
  ScheduleExpression: "rate(5 minutes)",
  State: "ENABLED",
};

export const run = async () => {
  try {
    const data = await cweClient.send(new PutRuleCommand(params));
    console.log("Success, scheduled rule created; Rule ARN:", data);
    return data; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
// Uncomment this line to run execution within this file.
// run();
```

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

```
node putRule.js
```

This example code can be found here on GitHub.

Adding a Lambda function target

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

```
import { CloudWatchEventsClient } from "@aws-sdk/client-cloudwatch-events";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon CloudWatch service client object.
export const cweClient = new CloudWatchEventsClient({ region: REGION });
```

This code is available here on GitHub.

Create a Node.js module with the file name `putTargets.js`. Be sure to configure the SDK as previously shown, including downloading the CloudWatch client. To access CloudWatch Events, create an `CloudWatchEvents` service object. Create a JSON object containing the parameters needed to specify the rule to which to attach the target, including the ARN of the Lambda function you created. Call the `PutTargetsCommand` method of the `CloudWatchEvents` service object.

> **Note**
> Replace *LAMBDA_FUNCTION_ARN* with the ARN of the Lambda function.

```
// Import required AWS SDK clients and commands for Node.js
import { PutTargetsCommand } from "@aws-sdk/client-cloudwatch-events";
import { cweClient } from "./libs/cloudWatchEventsClient.js";

// Set the parameters
export const params = {
  Rule: "DEMO_EVENT",
  Targets: [
    {
      Arn: "LAMBDA_FUNCTION_ARN", //LAMBDA_FUNCTION_ARN
      Id: "myCloudWatchEventsTarget",
    },
  ],
};

export const run = async () => {
  try {
    const data = await cweClient.send(new PutTargetsCommand(params));
    console.log("Success, target added; requestID: ", data);
    return data; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
// Uncomment this line to run execution within this file.
// run();
```

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

```
node putTargets.js
```

This example code can be found here on GitHub.

Sending events

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

```
import { CloudWatchEventsClient } from "@aws-sdk/client-cloudwatch-events";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon CloudWatch service client object.
export const cweClient = new CloudWatchEventsClient({ region: REGION });
```

This code is available here on GitHub.

Create a Node.js module with the file name `putEvents.js`. Be sure to configure the SDK as previously shown, including downloading the CloudWatch client. To access CloudWatch Events, create an `CloudWatchEvents` client service object. Create a JSON object containing the parameters needed to send events. For each event, include the source of the event, the ARNs of any resources affected by the event, and details for the event. Call the `PutEventsCommands` method of the `CloudWatchEvents` client service object.

Note
Replace *RESOURCE_ARN* with the resources affected by the event.

```
// Import required AWS SDK clients and commands for Node.js
import { PutEventsCommand } from "@aws-sdk/client-cloudwatch-events";
import { cweClient } from "./libs/cloudWatchEventsClient.js";

// Set the parameters
export const params = {
  Entries: [
    {
      Detail: '{ "key1": "value1", "key2": "value2" }',
      DetailType: "appRequestSubmitted",
      Resources: [
        "RESOURCE_ARN", //RESOURCE_ARN
      ],
      Source: "com.company.app",
    },
  ],
};

export const run = async () => {
  try {
    const data = await cweClient.send(new PutEventsCommand(params));
    console.log("Success, event sent; requestID:", data);
    return data; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
// Uncomment this line to run execution within this file.
// run();
```

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

```
node putEvents.js
```

This example code can be found here on GitHub.

Using subscription filters in Amazon CloudWatch Logs

This Node.js code example shows:

- How to create and delete filters for log events in CloudWatch Logs.

The scenario

Subscriptions provide access to a real-time feed of log events from CloudWatch Logs and deliver that feed to other services, such as an Amazon Kinesis stream or AWS Lambda, for custom processing, analysis, or loading to other systems. A subscription filter defines the pattern to use for filtering which log events are delivered to your AWS resource.

In this example, a series of Node.js modules are used to list, create, and delete a subscription filter in CloudWatch Logs. The destination for the log events is a Lambda function. The Node.js modules use the SDK for JavaScript to manage subscription filters using these methods of the `CloudWatchLogs` client class:

- `PutSubscriptionFilterCommand`
- `DescribeSubscriptionFilterCommand`
- `DeleteSubscriptionFilterCommand`

For more information about CloudWatch Logs subscriptions, see Real-time processing of log data with subscriptions in the *Amazon CloudWatch Logs User 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 Loading credentials in Node.js from the shared credentials file (p. 33).
- Create an AWS Lambda function as the destination for log events. You will need to use the Amazon Resource Name (ARN) of this function. For more information about setting up a Lambda function, see Subscription filters with Lambda in the *Amazon CloudWatch Logs User Guide*.
- Create an IAM role whose policy grants permission to invoke the Lambda function you created and grants full access to CloudWatch Logs or apply the following policy to the execution role you create for the Lambda function. For more information about creating an IAM role, see Creating a role to delegate permissions to an AWS service in the *IAM User Guide*.

Use the following role policy when creating the IAM role.

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

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

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 (p. 56).

Describing existing subscription filters

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

```
import { CloudWatchLogsClient } from "@aws-sdk/client-cloudwatch-logs";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon CloudWatch Logs service client object.
export const cwlClient = new CloudWatchLogsClient({ region: REGION });
```

This code is available here on GitHub.

Create a Node.js module with the file name `describeSubscriptionFilters.js`. Be sure to configure the SDK as previously shown, including downloading the CloudWatch client. Create a JSON object containing the parameters needed to describe your existing filters, including the name of the log group and the maximum number of filters to describe. Call the `DescribeSubscriptionFiltersCommand` method.

Note

Replace *GROUP_NAME* with the name of the group.

```

// Import required AWS SDK clients and commands for Node.js
import { DescribeSubscriptionFiltersCommand } from "@aws-sdk/client-cloudwatch-logs";
import { cwlClient } from "./libs/cloudWatchLogsClient.js";

// Set the parameters
```

```
export const params = {
  logGroupName: "GROUP_NAME", //GROUP_NAME
  limit: 5
};

export const run = async () => {
  try {
    const data = await cwlClient.send(
      new DescribeSubscriptionFiltersCommand(params)
    );
    console.log("Success", data.subscriptionFilters);
    return data; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
// Uncomment this line to run execution within this file.
// run();
```

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

```
node describeSubscriptionFilters.js
```

This example code can be found here on GitHub.

Creating a subscription filter

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

```
import { CloudWatchLogsClient } from "@aws-sdk/client-cloudwatch-logs";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon CloudWatch Logs service client object.
export const cwlClient = new CloudWatchLogsClient({ region: REGION });
```

This code is available here on GitHub.

Create a Node.js module with the file name `describeSubscriptionFilters.js`. Be sure to configure the SDK as previously shown, including downloading the CloudWatch client. Create a JSON object containing the parameters needed to create a filter, including the ARN of the destination Lambda function, the name of the filter, the string pattern for filtering, and the name of the log group. Call the `PutSubscriptionFiltersCommand` method.

> **Note**
> Replace *`LAMBDA_FUNCTION-ARN`* with the ARN of the Lambda function, *`FILTER_NAME`* with the name of the filter, and *`LOG_GROUP`* with the log group.

```


// Import required AWS SDK clients and commands for Node.js
import {
  PutSubscriptionFilterCommand,
} from "@aws-sdk/client-cloudwatch-logs";
import { cwlClient } from "./libs/cloudWatchLogsClient.js";

// Set the parameters
export const params = {
  destinationArn: "LAMBDA_FUNCTION_ARN", //LAMBDA_FUNCTION_ARN
```

```
  filterName: "FILTER_NAME", //FILTER_NAME
  filterPattern: "ERROR",
  logGroupName: "LOG_GROUP", //LOG_GROUP
};

export const run = async () => {
  try {
    const data = await cwlClient.send(new PutSubscriptionFilterCommand(params));
    console.log("Success", data.subscriptionFilters);
    return data; //For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
// Uncomment this line to run execution within this file.
// run();
```

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

```
node describeSubscriptionFilters.js
```

This example code can be found here on GitHub.

Deleting a subscription filter

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

```
import { CloudWatchLogsClient } from "@aws-sdk/client-cloudwatch-logs";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon CloudWatch Logs service client object.
export const cwlClient = new CloudWatchLogsClient({ region: REGION });
```

This code is available here on GitHub.

Create a Node.js module with the file name `deletesubScriptionFilters.js`. Be sure to configure the SDK as previously shown, including downloading the CloudWatch client. Create a JSON object containing the parameters needed to delete a filter, including the names of the filter and the log group. Call the `DeleteSubscriptionFiltersCommand` method.

> **Note**
> Replace *FILTER_NAME* with the name of the filter, and *LOG_GROUP* with the log group.

```
// Import required AWS SDK clients and commands for Node.js
import {
  DeleteSubscriptionFilterCommand
} from "@aws-sdk/client-cloudwatch-logs";
import { cwlClient } from "./libs/cloudWatchLogsClient.js";

// Set the parameters
export const params = {
  filterName: "FILTER", //FILTER
  logGroupName: "LOG_GROUP", //LOG_GROUP
};

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

```
    const data = await cwlClient.send(
      new DeleteSubscriptionFilterCommand(params)
    );
    console.log(
      "Success, subscription filter deleted",
      data
    );
    return data; //For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
// Uncomment this line to run execution within this file.
// run();
```

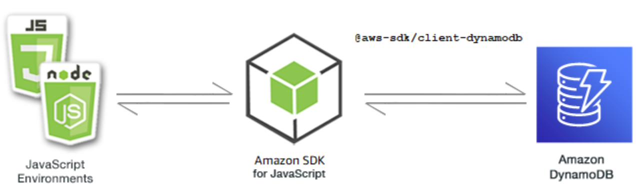
To run the example, enter the following at the command prompt.

```
node deletesubScriptionFilters.js
```

This example code can be found here on GitHub.

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

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 (p. 1).
- Create a shared configurations file with your user credentials. For more information about providing a shared credentials file, see Loading credentials in Node.js from the shared credentials file (p. 33).

> **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 sytax, please refer to JavaScript ES6/CommonJS syntax (p. 56)

Creating a table

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

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon DynamoDB service client object.
const ddbClient = new DynamoDBClient({ region: REGION });
export { ddbClient };
```

This code is available here on GitHub.

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

Note
Replace *TABLE_NAME*with the name of the table, *ATTRIBUTE_NAME_1* with the name of the partition key, *ATTRIBUTE_NAME_2* with the name of the sort key (optionally), and *ATTRIBUTE_TYPE* with the type of the attribute (for example, N [for a number], S [for a string] etc.).

Note
The primary key for the table is composed of the following attributes:

- `Season`
- `Episode`

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

// Set the parameters
export const params = {
  AttributeDefinitions: [
    {
      AttributeName: "Season", //ATTRIBUTE_NAME_1
      AttributeType: "N", //ATTRIBUTE_TYPE
    },
    {
      AttributeName: "Episode", //ATTRIBUTE_NAME_2
      AttributeType: "N", //ATTRIBUTE_TYPE
    },
  ],
  KeySchema: [
    {
      AttributeName: "Season", //ATTRIBUTE_NAME_1
      KeyType: "HASH",
    },
    {
      AttributeName: "Episode", //ATTRIBUTE_NAME_2
      KeyType: "RANGE",
    },
  ],
  ProvisionedThroughput: {
    ReadCapacityUnits: 1,
    WriteCapacityUnits: 1,
  },
  TableName: "TEST_TABLE", //TABLE_NAME
  StreamSpecification: {
    StreamEnabled: false,
  },
};

export const run = async () => {
  try {
    const data = await ddbClient.send(new CreateTableCommand(params));
    console.log("Table Created", data);
    return data;
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

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

```
node ddb_createtable.js
```

This example code can be found here on GitHub.

Listing your tables

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

```
// Create service client module using ES6 syntax.
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon DynamoDB service client object.
const ddbClient = new DynamoDBClient({ region: REGION });
export { ddbClient };
```

This code is available here on GitHub.

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

```

// Import required AWS SDK clients and commands for Node.js
import { ListTablesCommand } from "@aws-sdk/client-dynamodb";
import { ddbClient } from "./libs/ddbClient.js";

export const run = async () => {
  try {
    const data = await ddbClient.send(new ListTablesCommand({}));
    console.log(data);
    // console.log(data.TableNames.join("\n"));
    return data;
  } catch (err) {
    console.error(err);
  }
};
run();
```

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

```
node ddb_listtables.js
```

This example code can be found here on GitHub.

Describing a table

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

```
// Create service client module using ES6 syntax.
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
```

```
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon DynamoDB service client object.
const ddbClient = new DynamoDBClient({ region: REGION });
export { ddbClient };
```

This code is available here on GitHub.

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

> **Note**
> Replace *`TABLE_NAME`* with the name of the table.

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

// Set the parameters
export const params = { TableName: "TABLE_NAME" }; //TABLE_NAME

export const run = async () => {
  try {
    const data = await ddbClient.send(new DescribeTableCommand(params));
    console.log("Success", data);
    // console.log("Success", data.Table.KeySchema);
    return data;
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

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

```
node ddb_describetable.js
```

This example code can be found here on GitHub.

Deleting a table

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

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon DynamoDB service client object.
const ddbClient = new DynamoDBClient({ region: REGION });
export { ddbClient };
```

This code is available here on GitHub.

Create a Node.js module with the file name `ddb_deletetable.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. To access DynamoDB, create a `DynamoDB` client service object. Create a JSON object containing the parameters needed to delete a table, which in this example includes the name of the table provided as a command-line parameter. Call the `DeleteTableCommand` method of the DynamoDB service object.

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

// Set the parameters
export const params = {
  TableName: "CUSTOMER_LIST_NEW",
};

export const run = async () => {
  try {
    const data = await ddbClient.send(new DeleteTableCommand(params));
    console.log("Success, table deleted", data);
    return data;
  } catch (err) {
      console.log("Error", err);
    }
};
run();
```

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

```
node ddb_deletetable.js
```

This example code can be found here on GitHub.

Reading and writing a single item in DynamoDB

This Node.js code example shows:

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

The scenario

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

- PutItemCommand
- 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 Loading credentials in Node.js from the shared credentials file (p. 33).
- Create a DynamoDB table whose items you can access. For more information about creating a DynamoDB table, see Creating and using tables in DynamoDB (p. 78).

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 sytax, please refer to JavaScript ES6/CommonJS syntax (p. 56)

Writing an item

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

```
// Create service client module using ES6 syntax.
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon DynamoDB service client object.
const ddbClient = new DynamoDBClient({ region: REGION });
export { ddbClient };
```

This code is available here on GitHub.

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

Note
Replace ***TABLE_NAME*** with the name of the table.

Note
The following code example writes to a table with a primary key composed of only a partion key - `CUSTOMER_ID` - and a sort key - `CUSTOMER_NAME`. If the table's primary key is composed of only a partition key, you only specify the partition key.

```


// Import required AWS SDK clients and commands for Node.js
import { PutItemCommand } from "@aws-sdk/client-dynamodb";
import { ddbClient } from "./libs/ddbClient.js";

// Set the parameters
export const params = {
  TableName: "TABLE_NAME",
  Item: {
    CUSTOMER_ID: { N: "001" },
    CUSTOMER_NAME: { S: "Richard Roe" },
  },
};
```

```
export const run = async () => {
  try {
    const data = await ddbClient.send(new PutItemCommand(params));
    console.log(data);
    return data;
  } catch (err) {
    console.error(err);
  }
};
run();
```

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

```
node ddb_putitem.js
```

This example code can be found here on GitHub.

Getting an item

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

```
// Create service client module using ES6 syntax.
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon DynamoDB service client object.
const ddbClient = new DynamoDBClient({ region: REGION });
export { ddbClient };
```

This code is available here on GitHub.

Create a Node.js module with the file name ddb_getitem.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.

> **Note**
> Replace *TABLE_NAME* with the name of the table, *KEY_NAME* with the primary key of the table, *KEY_NAME_VALUE* with the value of the primary key row containing the attribute value, and *ATTRIBUTE_NAME* the name of the attribute column containing the attribute value.

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

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

// Set the parameters
```

```
export const params = {
  TableName: "TABLE_NAME", //TABLE_NAME
  Key: {
    KEY_NAME: { N: "KEY_VALUE" },
  },
  ProjectionExpression: "ATTRIBUTE_NAME",
};

export const run = async () => {
  const data = await ddbClient.send(new GetItemCommand(params));
  return data;
  console.log("Success", data.Item);
};
run();
```

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

```
node ddb_getitem.js
```

This example code can be found here on GitHub.

Deleting an item

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

```
// Create service client module using ES6 syntax.
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon DynamoDB service client object.
const ddbClient = new DynamoDBClient({ region: REGION });
export { ddbClient };
```

This code is available here on GitHub.

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

Note
Replace ***TABLE_NAME*** with the name of the table.

Note
The following code example below deletes a item with a primary key composed of only a partion key - `KEY_NAME` - and not of both a partion and sort key. If the table has a primary key composed of a partition key and a sort key, you must also specify the sort key name and attribute.

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

// Set the parameters
export const params = {
  TableName: "CUSTOMER_LIST_NEWEST",
  Key: {
```

```
    CUSTOMER_ID: { N: "1" },
  },
};

export const run = async () => {
  try {
    const data = await ddbClient.send(new DeleteItemCommand(params));
    console.log("Success, item deleted", data);
    return data;
  } catch (err) {
    console.log("Error", err);
    /*if (err && err.code === "ResourceNotFoundException") {
      console.log("Error: Table not found");
    } else if (err && err.code === "ResourceInUseException") {
      console.log("Error: Table in use");
    }*/
  }
};
run();
```

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

```
node ddb_deleteitem.js
```

This example code can be found here on GitHub.

Reading and writing items in batch in DynamoDB

This Node.js code example shows:

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

The scenario

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

- BatchGetItemCommand
- BatchWriteItemCommand

Prerequisite tasks

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

- Set up the project environment to run these Node JypeScript 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 Loading credentials in Node.js from the shared credentials file (p. 33).

- Create a DynamoDB table whose items you can access. For more information about creating a DynamoDB table, see Creating and using tables in DynamoDB (p. 78).

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 sytax, please refer to JavaScript ES6/CommonJS syntax (p. 56)

Reading items in Batch

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

```
// Create service client module using ES6 syntax.
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon DynamoDB service client object.
const ddbClient = new DynamoDBClient({ region: REGION });
export { ddbClient };
```

This code is available here on GitHub.

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

Note
Replace ***TABLE_NAME*** with the name of the table, ***KEY_NAME*** with the primary key of the table, ***KEY_VALUE*** with the value of the primary key row containing the attribute value, and ***ATTRIBUTE_NAME*** the name of the attribute column containing the attribute value.

Note
This the following code below batch retrieves items from a table with a primary key composed of only a partion key - `KEY_NAME` - and not of both a partion 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 for each item.

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

// Set the parameters
export const params = {
  RequestItems: {
    TABLE_NAME: {
      Keys: [
        {
          KEY_NAME_1: { N: "KEY_VALUE" },
          KEY_NAME_2: { N: "KEY_VALUE" },
          KEY_NAME_3: { N: "KEY_VALUE" },
        },
      ],
      ProjectionExpression: "ATTRIBUTE_NAME",
```

```
    },
  },
};

export const run = async () => {
  try {
    const data = await ddbClient.send(new BatchGetItemCommand(params));
    console.log("Success, items retrieved", data);
    return data;
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

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

```
node ddb_batchgetitem.js
```

This example code can be found here on GitHub.

Writing items in Batch

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

```
// Create service client module using ES6 syntax.
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon DynamoDB service client object.
const ddbClient = new DynamoDBClient({ region: REGION });
export { ddbClient };
```

This code is available here on GitHub.

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

> **Note**
> Replace *`TABLE_NAME`* with the name of the table, *`KEYS`* with the primary key of the item, *`KEY_VALUE`* with the value of the primary key row containing the attribute value, and *`ATTRIBUTE_NAME`* the name of the attribute column containing the attribute value.

The following code example batch writes items to a table with a primary key composed of only a partion key - `KEY_NAME` - and not of both a partion 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 for each item.

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

// Set the parameters
```

```
export const 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" },
          },
        },
      },
    ],
  },
};

export const run = async () => {
  try {
    const data = await ddbClient.send(new BatchWriteItemCommand(params));
    console.log("Success, items inserted", data);
    return data;
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

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

```
node ddb_batchwriteitem.js
```

This example code can be found here on GitHub.

Querying and scanning a DynamoDB table

This Node.js code example shows:

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

The scenario

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

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

- QueryCommand
- ScanCommand

Prerequisite tasks

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

- Set up the project environment to run these Node.js examples, and install the required AWS SDK for JavaScript and third-party modules. Follow the instructions on GitHub.
- Create a shared configurations file with your user credentials. For more information about providing a shared credentials file, see Loading credentials in Node.js from the shared credentials file (p. 33).
- Create a DynamoDB table whose items you can access. For more information about creating a DynamoDB table, see Creating and using tables in DynamoDB (p. 78).

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 sytax, please refer to JavaScript ES6/CommonJS syntax (p. 56)

Querying a table

This example queries a table that contains episode information about a video series, returning the episode titles and subtitles of second season episodes past episode 9 that contain a specified phrase in their subtitle.

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

```
// Create service client module using ES6 syntax.
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon DynamoDB service client object.
const ddbClient = new DynamoDBClient({ region: REGION });
export { ddbClient };
```

This code is available here on GitHub.

Create a Node.js module with the file name `ddb_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 servicec 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.

The primary key for the table is composed of the following attributes:

- `Season`
- `Episode`

You can run the code here on GitHub to create the table that this query targets, and the code here on GitHub to populate the table.

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

// Set the parameters
export const params = {
  KeyConditionExpression: "Season = :s and Episode > :e",
  FilterExpression: "contains (Subtitle, :topic)",
  ExpressionAttributeValues: {
    ":s": { N: "1" },
    ":e": { N: "2" },
    ":topic": { S: "SubTitle" },
  },
  ProjectionExpression: "Episode, Title, Subtitle",
  TableName: "EPISODES_TABLE",
};

export const run = async () => {
  try {
    const data = await ddbClient.send(new QueryCommand(params));
    return data;
    data.Items.forEach(function (element, index, array) {
      console.log(element.Title.S + " (" + element.Subtitle.S + ")");
    });
  } catch (err) {
    console.error(err);
  }
};
run();
```

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

```
node ddb_query.js
```

This example code can be found here on GitHub.

Scanning a table

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

```
// Create service client module using ES6 syntax.
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon DynamoDB service client object.
const ddbClient = new DynamoDBClient({ region: REGION });
export { ddbClient };
```

This code is available here on GitHub.

Create a Node.js module with the file name `ddb_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 `ScanQuery` method of the DynamoDB service object.

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

// Set the parameters.
export const params = {
  // Specify which items in the results are returned.
  FilterExpression: "Subtitle = :topic AND Season = :s AND Episode = :e",
  // Define the expression attribute value, which are substitutes for the values you want
 to compare.
  ExpressionAttributeValues: {
    ":topic": { S: "SubTitle2" },
    ":s": { N: "1" },
    ":e": { N: "2" },
  },
  // Set the projection expression, which the the attributes that you want.
  ProjectionExpression: "Season, Episode, Title, Subtitle",
  TableName: "EPISODES_TABLE",
};


export const run = async () => {
  try {
    const data = await ddbClient.send(new ScanCommand(params));
    return data;
    data.Items.forEach(function (element, index, array) {
      console.log(element.Title.S + " (" + element.Subtitle.S + ")");
    });
  } catch (err) {
    console.log("Error", err);
  }
}
run();
```

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

```
node ddb_scan.js
```

This example code can be found here on GitHub.

Using the DynamoDB Document Client

This Node.js code example shows:

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

The Scenario

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

For more information about the DynamoDB Document Client, see @aws-sdk/lib-dynamodb README on GitHub. For more information about programming with Amazon DynamoDB, see Programming with DynamoDB 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 class:

- getItem
- putItem
- updateItem
- query
- deleteItem

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 Loading credentials in Node.js from the shared credentials file (p. 33).
- 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 (p. 78). 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 sytax, please refer to JavaScript ES6/CommonJS syntax (p. 56)

Getting an Item from a Table

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

```
// Create service client module using ES6 syntax.
import { DynamoDBDocumentClient} from "@aws-sdk/lib-dynamodb";
import {ddbClient} from "./ddbClient";
// Set the AWS Region.
```

```
const REGION = "REGION"; //e.g. "us-east-1"

const marshallOptions = {
    // Whether to automatically convert empty strings, blobs, and sets to `null`.
    convertEmptyValues: false, // false, by default.
    // Whether to remove undefined values while marshalling.
    removeUndefinedValues: false, // false, by default.
    // Whether to convert typeof object to map attribute.
    convertClassInstanceToMap: false, // false, by default.
};

const unmarshallOptions = {
    // Whether to return numbers as a string instead of converting them to native
 JavaScript numbers.
    wrapNumbers: false, // false, by default.
};

const translateConfig = { marshallOptions, unmarshallOptions };

// Create the DynamoDB Document client.
const ddbDocClient = DynamoDBDocumentClient.from(ddbClient, translateConfig);

export { ddbDocClient };
```

This code is available here on GitHub.

In the `libs` directory create a Node.js module with the file name `ddbClient.js`. Copy and paste the code below into it, which creates the DynamoDB client object. Replace *`REGION`* with your AWS region.

```
// Create service client module using ES6 syntax.
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon DynamoDB service client object.
const ddbClient = new DynamoDBClient({ region: REGION });
export { ddbClient };
```

This code is available here on GitHub.

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

```


import { GetCommand } from "@aws-sdk/lib-dynamodb";
import { ddbDocClient } from "./libs/ddbDocClient";

// Set the parameters.
export const params = {
  TableName: "TABLE_NAME",
  /*
  Convert the key JavaScript object you are retrieving to the
  required Amazon DynamoDB record. The format of values specifies
  the datatype. The following list demonstrates different
  datatype formatting requirements:
  String: "String",
  NumAttribute: 1,
```

```
  BoolAttribute: true,
  ListAttribute: [1, "two", false],
  MapAttribute: { foo: "bar" },
  NullAttribute: null
   */
  Key: {
    primaryKey: "VALUE", // For example, 'Season': 2.
    sortKey: "VALUE", // For example,  'Episode': 1; (only required if table has sort key).
  },
};

export const run = async () => {
  try {
    const data = await ddbDocClient.send(new GetCommand(params));
    console.log("Success :", data);
    // console.log("Success :", data.Item);
    return data;
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

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

```
node ddbdoc_get_item.js // To use JavaScript, enter 'node ddbdoc_get_item.js
```

This example code can be found here on GitHub.

Putting an Item in a Table

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

```
// Create service client module using ES6 syntax.
import { DynamoDBDocumentClient} from "@aws-sdk/lib-dynamodb";
import {ddbClient} from "./ddbClient";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"

const marshallOptions = {
    // Whether to automatically convert empty strings, blobs, and sets to `null`.
    convertEmptyValues: false, // false, by default.
    // Whether to remove undefined values while marshalling.
    removeUndefinedValues: false, // false, by default.
    // Whether to convert typeof object to map attribute.
    convertClassInstanceToMap: false, // false, by default.
};

const unmarshallOptions = {
    // Whether to return numbers as a string instead of converting them to native
 JavaScript numbers.
    wrapNumbers: false, // false, by default.
};

const translateConfig = { marshallOptions, unmarshallOptions };

// Create the DynamoDB Document client.
const ddbDocClient = DynamoDBDocumentClient.from(ddbClient, translateConfig);
```

```
export { ddbDocClient };
```

This code is available here on GitHub.

In the libs directory create a Node.js module with the file name `ddbClient.js`. Copy and paste the code below into it, which creates the DynamoDB client object. Replace *REGION* with your AWS region.

```
// Create service client module using ES6 syntax.
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon DynamoDB service client object.
const ddbClient = new DynamoDBClient({ region: REGION });
export { ddbClient };
```

This code is available here on GitHub.

Create a Node.js module with the file name `ddbdoc_put_item.js`. Be sure to configure the SDK as previously shown, including installing the required clients and packages. This includes the `@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 { PutCommand } from "@aws-sdk/lib-dynamodb";
import { ddbDocClient } from "./libs/ddbDocClient";

// Set the parameters.
export const params = {
  TableName: "TABLE_NAME",
  /*
    Convert the key JavaScript object you are adding to the
    required Amazon DynamoDB record. The format of values specifies
    the datatype. The following list demonstrates different
    datatype formatting requirements:
    String: "String",
    NumAttribute: 1,
    BoolAttribute: true,
    ListAttribute: [1, "two", false],
    MapAttribute: { foo: "bar" },
    NullAttribute: null
     */
  Item: {
    primaryKey: "VALUE_1", // For example, 'Season': 2
    sortKey: "VALUE_2", // For example,  'Episode': 2 (only required if table has sort key)
    NEW_ATTRIBUTE_1: "NEW_ATTRIBUTE_1_VALUE", //For example 'Title': 'The Beginning'
  },
};

export const run = async () => {
  try {
    const data = await ddbDocClient.send(new PutCommand(params));
    console.log("Success - item added or updated", data);
    return data;
  } catch (err) {
    console.log("Error", err);
  }
};
```

```
run();
```

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

```
node ddbdoc_put_item.js // To use JavaScript, enter 'node ddbdoc_put_item.js'
```

This example code can be found here on GitHub.

Updating an Item in a Table

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

```
// Create service client module using ES6 syntax.
import { DynamoDBDocumentClient} from "@aws-sdk/lib-dynamodb";
import {ddbClient} from "./ddbClient";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"

const marshallOptions = {
    // Whether to automatically convert empty strings, blobs, and sets to `null`.
    convertEmptyValues: false, // false, by default.
    // Whether to remove undefined values while marshalling.
    removeUndefinedValues: false, // false, by default.
    // Whether to convert typeof object to map attribute.
    convertClassInstanceToMap: false, // false, by default.
};

const unmarshallOptions = {
    // Whether to return numbers as a string instead of converting them to native
 JavaScript numbers.
    wrapNumbers: false, // false, by default.
};

const translateConfig = { marshallOptions, unmarshallOptions };

// Create the DynamoDB Document client.
const ddbDocClient = DynamoDBDocumentClient.from(ddbClient, translateConfig);

export { ddbDocClient };
```

This code is available here on GitHub.

In the `libs` directory create a Node.js module with the file name `ddbClient.js`. Copy and paste the code below into it, which creates the DynamoDB client object. Replace *REGION* with your AWS region.

```
// Create service client module using ES6 syntax.
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon DynamoDB service client object.
const ddbClient = new DynamoDBClient({ region: REGION });
export { ddbClient };
```

This code is available here on GitHub.

Create a Node.js module with the file name `ddbdoc_update_item.js`. Be sure to configure the SDK as previously shown, including installing the required clients and packages. This includes the `@aws-sdk/lib-dynamodb`, a library package that provides document client functionality to `@aws-sdk/client-`

dynamodb. Next, set the configuration as shown below for marshalling and unmarshalling - as an optional second parameter - during creation of document client. Next, create the clients. Create a JSON object containing the parameters needed to write an item to the table, which in this example includes the name of the table, the key of the item to update, a set of UpdateExpressions that define the attributes of the item to update with tokens you assign values to in the ExpressionAttributeValues parameters.Call the UpdateCommand method of the DynamoDB Document client.

```
import { UpdateCommand } from "@aws-sdk/lib-dynamodb";
import { ddbDocClient } from "./libs/ddbDocClient";

// Set the parameters
export const params = {
  TableName: "TABLE_NAME",
  /*
  Convert the attribute JavaScript object you are updating to the required
  Amazon  DynamoDB record. The format of values specifies the datatype. The
  following list demonstrates different datatype formatting requirements:
  String: "String",
  NumAttribute: 1,
  BoolAttribute: true,
  ListAttribute: [1, "two", false],
  MapAttribute: { foo: "bar" },
  NullAttribute: null
   */
  Key: {
    primaryKey: "VALUE_1", // For example, 'Season': 2.
    sortKey: "VALUE_2", // For example,  'Episode': 1; (only required if table has sort
 key).
  },
  // Define expressions for the new or updated attributes
  UpdateExpression: "set ATTRIBUTE_NAME_1 = :t, ATTRIBUTE_NAME_2 = :s", // For example,
 "'set Title = :t, Subtitle = :s'"
  ExpressionAttributeValues: {
    ":t": "NEW_ATTRIBUTE_VALUE_1", // For example ':t' : 'NEW_TITLE'
    ":s": "NEW_ATTRIBUTE_VALUE_2", // For example ':s' : 'NEW_SUBTITLE'
  },
  ReturnValues: "ALL_NEW"
};

export const run = async () => {
  try {
    const data = await ddbDocClient.send(new UpdateCommand(params));
    console.log("Success - item added or updated", data);
    return data;
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

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

```
node ddbdoc_update_item.js
```

This example code can be found here on GitHub.

Querying a Table

Create a libs directory, and create a Node.js module with the file name ddbDocClient.js. Copy and paste the code below into it, which creates the DynamoDB document client object. Replace *REGION* with your AWS region.

```
// Create service client module using ES6 syntax.
import { DynamoDBDocumentClient} from "@aws-sdk/lib-dynamodb";
import {ddbClient} from "./ddbClient";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"

const marshallOptions = {
    // Whether to automatically convert empty strings, blobs, and sets to `null`.
    convertEmptyValues: false, // false, by default.
    // Whether to remove undefined values while marshalling.
    removeUndefinedValues: false, // false, by default.
    // Whether to convert typeof object to map attribute.
    convertClassInstanceToMap: false, // false, by default.
};

const unmarshallOptions = {
    // Whether to return numbers as a string instead of converting them to native
 JavaScript numbers.
    wrapNumbers: false, // false, by default.
};

const translateConfig = { marshallOptions, unmarshallOptions };

// Create the DynamoDB Document client.
const ddbDocClient = DynamoDBDocumentClient.from(ddbClient, translateConfig);

export { ddbDocClient };
```

This code is available here on GitHub.

In the `libs` directory create a Node.js module with the file name `ddbClient.js`. Copy and paste the code below into it, which creates the DynamoDB client object. Replace *`REGION`* with your AWS region.

```
// Create service client module using ES6 syntax.
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon DynamoDB service client object.
const ddbClient = new DynamoDBClient({ region: REGION });
export { ddbClient };
```

This code is available here on GitHub.

This example queries a table that contains episode information about a video series, returning the episode titles and subtitles of second season episodes past episode 9 that contain a specified phrase in their subtitle.

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

```
import { QueryCommand } from "@aws-sdk/lib-dynamodb";
import { ddbDocClient } from "./libs/ddbDocClient";

// Set the parameters
export const params = {
  TableName: "TABLE_NAME",
```

```
  /*
  Convert the JavaScript object defining the objects to the required
  Amazon DynamoDB record. The format of values specifies the datatype. The
  following list demonstrates different datatype formatting requirements:
  String: "String",
  NumAttribute: 1,
  BoolAttribute: true,
  ListAttribute: [1, "two", false],
  MapAttribute: { foo: "bar" },
  NullAttribute: null
   */
  ExpressionAttributeValues: {
    ":s": 1,
    ":e": 1,
    ":topic": "Title2",
  },
  // Specifies the values that define the range of the retrieved items. In this case, items
 in Season 2 before episode 9.
  KeyConditionExpression: "Season = :s and Episode > :e",
  // Filter that returns only episodes that meet previous criteria and have the subtitle
 'The Return'
  FilterExpression: "contains (Subtitle, :topic)",
};

export const run = async () => {
  try {
    const data = await ddbDocClient.send(new QueryCommand(params));
    console.log("Success. Item details: ", data);
    // console.log("Success. Item details: ", data.Items);
    return data;
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

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

```
node ddbdoc_query_item.js
```

This example code can be found here on GitHub.

Deleting an Item from a Table

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

```
// Create service client module using ES6 syntax.
import { DynamoDBDocumentClient} from "@aws-sdk/lib-dynamodb";
import {ddbClient} from "./ddbClient";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"

const marshallOptions = {
    // Whether to automatically convert empty strings, blobs, and sets to `null`.
    convertEmptyValues: false, // false, by default.
    // Whether to remove undefined values while marshalling.
    removeUndefinedValues: false, // false, by default.
    // Whether to convert typeof object to map attribute.
    convertClassInstanceToMap: false, // false, by default.
};
```

```
const unmarshallOptions = {
    // Whether to return numbers as a string instead of converting them to native
 JavaScript numbers.
    wrapNumbers: false, // false, by default.
};

const translateConfig = { marshallOptions, unmarshallOptions };

// Create the DynamoDB Document client.
const ddbDocClient = DynamoDBDocumentClient.from(ddbClient, translateConfig);

export { ddbDocClient };
```

This code is available here on GitHub.

In the `libs` directory create a Node.js module with the file name `ddbClient.js`. Copy and paste the code below into it, which creates the DynamoDB client object. Replace ***`REGION`*** with your AWS region.

```
// Create service client module using ES6 syntax.
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon DynamoDB service client object.
const ddbClient = new DynamoDBClient({ region: REGION });
export { ddbClient };
```

This code is available here on GitHub.

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

```
import { DeleteCommand } from "@aws-sdk/lib-dynamodb";
import { ddbDocClient } from "./libs/ddbDocClient";

// Set the parameters
export const params = {
  TableName: "TABLE_NAME",
  /*
  Convert the key JavaScript object you are deleting to the
  required Amazon DynamoDB record. The format of values specifies
  the datatype. The following list demonstrates different
  datatype formatting requirements:
  String: "String",
  NumAttribute: 1,
  BoolAttribute: true,
  ListAttribute: [1, "two", false],
  MapAttribute: { foo: "bar" },
  NullAttribute: null
   */
  Key: {
    primaryKey: "VALUE_1", // For example, 'Season': 2.
    sortKey: "VALUE_2", // For example,  'Episode': 1; (only required if table has sort
 key).
  },
```

```
};

export const run = async () => {
  try {
    const data = await ddbDocClient.send(new DeleteCommand(params));
    console.log("Success - item deleted");
    return data;
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

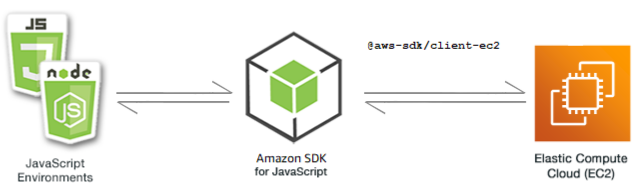
To run the example, enter the following at the command prompt.

```
node ddbdoc_delete_item.js
```

This example code can be found here on GitHub.

Amazon EC2 examples

Amazon Elastic Compute Cloud (Amazon EC2) is a web service that provides virtual server hosting in the cloud. It is designed to make web-scale cloud computing easier for developers by providing resizeable compute capacity.



The JavaScript API for Amazon EC2 is exposed through the `EC2` client class. For more information about using the Amazon EC2 client class, see Class: EC2 in the API Reference.

Topics

Creating an Amazon EC2 instance

This Node.js code example shows:

- How to create an Amazon EC2 instance from a public Amazon Machine Image (AMI).
- How to create and assign tags to the new Amazon EC2 instance.

About the example

In this example, you use a Node.js module to create an Amazon EC2 instance and assign both a key pair and tags to it. The code uses the SDK for JavaScript to create and tag an instance by using these methods of the Amazon EC2 client class:

- `RunInstancesCommand`
- `CreateTagsCommand`

Prerequisite tasks

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

- Create a shared configurations file with your user credentials. For more information about providing a shared credentials file, see Loading credentials in Node.js from the shared credentials file (p. 33).
- Create a key pair. For details, see Working with Amazon EC2 key pairs (p. 110). You use the name of the key pair in this example.

> **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 sytax, please refer to JavaScript ES6/CommonJS syntax (p. 56)

Creating and tagging an instance

Create a `libs` directory, and create a Node.js module with the file name `ec2Client.js`. Copy and paste the code below into it, which creates the Amazon EC2 client object. Replace *`REGION`* with your AWS Region.

```
const  { EC2Client } = require( "@aws-sdk/client-ec2");
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create anAmazon EC2 service client object.
const ec2Client = new EC2Client({ region: REGION });
module.exports = { ec2Client };
```

This example code can be found here on GitHub.

Create a Node.js module with the file name `ec2_createinstances.js`. Be sure to configure the SDK as previously shown, including installing the required clients and packages.

Create an object to pass the parameters for the `RunInstancesCommand` method of the `EC2` client class, including the name of the key pair to assign and the ID of the AMI to run. To call the `RunInstancesCommand` method, create an asynchronous function for invoking an Amazon EC2 client service object, passing the parameters.

The code next adds a `Name` tag to a new instance, which the Amazon EC2 console recognizes and displays in the **Name** field of the instance list. You can add up to 50 tags to an instance, all of which can be added in a single call to the `CreateTagsCommand` method.

Note

Replace *AMI_ID* with the ID of the Amazon Machine Image (AMI) to run, and *KEY_PAIR_NAME* of the key pair to assign to the AMI ID.

```
// Import required AWS SDK clients and commands for Node.js
const {
    CreateTagsCommand,
    RunInstancesCommand
} = require("@aws-sdk/client-ec2");
import { ec2Client } from "./libs/ec2Client";

// Set the parameters
const instanceParams = {
    ImageId: "AMI_ID", //AMI_ID
    InstanceType: "t2.micro",
    KeyName: "KEY_PAIR_NAME", //KEY_PAIR_NAME
    MinCount: 1,
    MaxCount: 1,
};

const run = async () => {
    try {
        const data = await ec2Client.send(new RunInstancesCommand(instanceParams));
        console.log(data.Instances[0].InstanceId);
        const instanceId = data.Instances[0].InstanceId;
        console.log("Created instance", instanceId);
        // Add tags to the instance
        const tagParams = {
            Resources: [instanceId],
            Tags: [
                {
                    Key: "Name",
                    Value: "SDK Sample",
                },
            ],
        };
        try {
            const data = await ec2Client.send(new CreateTagsCommand(tagParams));
            console.log("Instance tagged");
        } catch (err) {
            console.log("Error", err);
        }
    } catch (err) {
        console.log("Error", err);
    }
};
run();
```

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

```
node ec2_createinstances.js
```

This example code can be found here on GitHub.

Managing Amazon EC2 instances

This Node.js code example shows:

- How to retrieve basic information about your Amazon EC2 instances.
- How to start and stop detailed monitoring of an Amazon EC2 instance.
- How to start and stop an Amazon EC2 instance.
- How to reboot an Amazon EC2 instance.

The scenario

In this example, you use a series of Node.js modules to perform several basic instance management operations. The Node.js modules use the SDK for JavaScript to manage instances by using these Amazon EC2 client class methods:

- `DescribeInstancesCommand`
- `MonitorInstancesCommand`
- `UnmonitorInstancesCommand`
- `StartInstancesCommand`
- `StopInstancesCommand`
- `RebootInstancesCommand`

For more information about the lifecycle of Amazon EC2 instances, see Instance lifecycle in the *Amazon EC2 User Guide for Linux Instances*.

Prerequisite tasks

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

- Create a shared configurations file with your user credentials. For more information about providing a shared credentials file, see Loading credentials in Node.js from the shared credentials file (p. 33).
- Create an Amazon EC2 instance. For more information about creating Amazon EC2 instances, see Amazon EC2 instances in the *Amazon EC2 User Guide for Linux Instances* or Amazon EC2 instances in the *Amazon EC2 User Guide for Windows Instances*.

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 sytax, please refer to JavaScript ES6/CommonJS syntax (p. 56)

Describing your instances

Create a `libs` directory, and create a Node.js module with the file name `ec2Client.js`. Copy and paste the code below into it, which creates the Amazon EC2 client object. Replace ***`REGION`*** with your AWS Region.

```
const  { EC2Client } = require( "@aws-sdk/client-ec2");
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create anAmazon EC2 service client object.
const ec2Client = new EC2Client({ region: REGION });
module.exports = { ec2Client };
```

This example code can be found here on GitHub.

Create a Node.js module with the file name `ec2_describeinstances.js`. Be sure to configure the SDK as previously shown. Call the `DescribeInstancesCommand` method of the Amazon EC2 service object to retrieve a detailed description of your instances.

```
// Import required AWS SDK clients and commands for Node.js
import { DescribeInstancesCommand } from "@aws-sdk/client-ec2";
import { ec2Client } from "./libs/ec2Client";
const run = async () => {
  try {
    const data = await ec2Client.send(new DescribeInstancesCommand({}));
    console.log("Success", JSON.stringify(data));
    return data;
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

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

```
node ec2_describeinstances.js
```

This example code can be found here on GitHub.

Managing instance monitoring

Create a `libs` directory, and create a Node.js module with the file name `ec2Client.js`. Copy and paste the code below into it, which creates the Amazon EC2 client object. Replace *`REGION`* with your AWS Region.

```
const  { EC2Client } = require( "@aws-sdk/client-ec2");
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create anAmazon EC2 service client object.
const ec2Client = new EC2Client({ region: REGION });
module.exports = { ec2Client };
```

This example code can be found here on GitHub.

Create a Node.js module with the file name `ec2_monitorinstances.js`. Be sure to configure the SDK as previously shown. Add the instance IDs of the instances for which you want to control monitoring.

Based on the value of a command-line argument (`ON` or `OFF`), call either the `MonitorInstancesCommand` method of the Amazon EC2 service object to begin detailed monitoring of the specified instances or call the `UnmonitorInstancesCommand` method.

> **Note**
> Replace *`INSTANCE_ID`* with the IDs of the instances for which you want to control monitoring, and *`STATE`* with either `ON` or `OFF`.

```
// Import required AWS SDK clients and commands for Node.js
import {
  MonitorInstancesCommand,
  UnmonitorInstancesCommand,
} from "@aws-sdk/client-ec2";
```

```
import { ec2Client } from "./libs/ec2Client";

// Set the parameters
const params = { InstanceIds: ["INSTANCE_ID"] }; // Array of INSTANCE_IDs
const state = "STATE"; // STATE; i.e., 'ON' or 'OFF'

const run = async () => {
  if (process.argv[4].toUpperCase() === "ON") {
    try {
      const data = await ec2Client.send(new MonitorInstancesCommand(params));
      console.log("Success", data.InstanceMonitorings);
      return data;
    } catch (err) {
      console.log("Error", err);
    }
  } else if (process.argv[4].toUpperCase() === "OFF") {
    try {
      const data = await ec2Client.send(new UnmonitorInstancesCommand(params));
      console.log("Success", data.InstanceMonitorings);
      return data;
    } catch (err) {
      console.log("Error", err);
    }
  }
};
run();
```

To run the example, enter the following at the command prompt, specifying `ON` to begin detailed monitoring or `OFF` to discontinue monitoring.

```
node ec2_monitorinstances.js ON
```

This example code can be found here on GitHub.

Starting and stopping instances

Create a `libs` directory, and create a Node.js module with the file name `ec2Client.js`. Copy and paste the code below into it, which creates the Amazon EC2 client object. Replace *`REGION`* with your AWS Region.

```
const  { EC2Client } = require( "@aws-sdk/client-ec2");
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create anAmazon EC2 service client object.
const ec2Client = new EC2Client({ region: REGION });
module.exports = { ec2Client };
```

This example code can be found here on GitHub.

Create a Node.js module with the file name `ec2_startstopinstances.js`. Be sure to configure the SDK as previously shown. Add the instance IDs of the instances you want to start or stop.

Based on the value of a command-line argument (`START` or `STOP`), call either the `StartInstancesCommand` method of the Amazon EC2 service object to start the specified instances, or the `StopInstancesCommand` method to stop them.

> **Note**
> Replace *`INSTANCE_ID`* with the instance IDs of the instances you want to start or stop, and *`STATE`* with `START` or `STOP`.

```
// Import required AWS SDK clients and commands for Node.js.
import {
  StartInstancesCommand,
  StopInstancesCommand,
} from "@aws-sdk/client-ec2";
import { ec2Client } from "./libs/ec2Client";

// Set the parameters
const params = { InstanceIds: ["INSTANCE_ID"] }; // Array of INSTANCE_IDs
const command = "STATE"; // STATE i.e. "START" or "STOP"

const run = async () => {
  if (command.toUpperCase() === "START") {
    try {
      const data = await ec2Client.send(new StartInstancesCommand(params));
      console.log("Success", data.StartingInstances);
      return data;
    } catch (err) {
      console.log("Error2", err);
    }
  } else if (process.argv[2].toUpperCase() === "STOP") {
    try {
      const data = await ec2Client.send(new StopInstancesCommand(params));
      console.log("Success", data.StoppingInstances);
      return data;
    } catch (err) {
      console.log("Error", err);
    }
  }
};
run();
```

To run the example, enter the following at the command prompt specifying `START` to start the instances or `STOP` to stop them.

```
node ec2_startstopinstances.js
```

This example code can be found here on GitHub.

Rebooting instances

Create a `libs` directory, and create a Node.js module with the file name `ec2Client.js`. Copy and paste the code below into it, which creates the Amazon EC2 client object. Replace ***`REGION`*** with your AWS Region.

```
const  { EC2Client } = require( "@aws-sdk/client-ec2");
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create anAmazon EC2 service client object.
const ec2Client = new EC2Client({ region: REGION });
module.exports = { ec2Client };
```

This example code can be found here on GitHub.

Create a Node.js module with the file name `ec2_rebootinstances.js`. Be sure to configure the SDK as previously shown. Add the instance IDs of the instances you want to reboot. Call the `RebootInstancesCommand` method of the `EC2` client service object to reboot the specified instances.

Note

Replace `INSTANCE_ID` with the IDs of the instance you want to reboot.

```
// Import required AWS SDK clients and commands for Node.js
import { RebootInstancesCommand } from "@aws-sdk/client-ec2";
import { ec2Client } from "./libs/ec2Client.js";

// Set the parameters
const params = { InstanceIds: ["INSTANCE_ID"] }; // Array of INSTANCE_IDs

const run = async () => {
  try {
    const data = await ec2Client.send(new RebootInstancesCommand(params));
    console.log("Success", data.InstanceMonitorings);
    return data;
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

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

```
node ec2_rebootinstances.js
```

This example code can be found here on GitHub.

Working with Amazon EC2 key pairs

This Node.js code example shows:

- How to retrieve information about your key pairs.
- How to create a key pair to access an Amazon EC2 instance.
- How to delete an existing key pair.

The scenario

Amazon EC2 uses public–key cryptography to encrypt and decrypt login information. Public–key cryptography uses a public key to encrypt data, then the recipient uses the private key to decrypt the data. The public and private keys are known as a *key pair*.

In this example, you use a series of Node.js modules to perform several Amazon EC2 key pair management operations. The Node.js modules use the SDK for JavaScript to manage instances by using these methods of the Amazon EC2 client class:

- `CreateKeyPairCommand`
- `DeleteKeyPairCommand`
- `DescribeKeyPairsCommand`

For more information about the Amazon EC2 key pairs, see Amazon EC2 key pairs in the *Amazon EC2 User Guide for Linux Instances* or Amazon EC2 key pairs and Windows Instances in the *Amazon EC2 User Guide for Windows Instances*.

Prerequisite tasks

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

- Install SDK for JavaScript Amazon EC2 client. For more information, see What's new in Version 3 (p. 1).
- Create a shared configurations file with your user credentials. For more information about providing a shared credentials file, see Loading credentials in Node.js from the shared credentials file (p. 33).

> **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 sytax, please refer to JavaScript ES6/CommonJS syntax (p. 56)

Describing your key pairs

Create a `libs` directory, and create a Node.js module with the file name `ec2Client.js`. Copy and paste the code below into it, which creates the Amazon EC2 client object. Replace *`REGION`* with your AWS Region.

```
const  { EC2Client } = require( "@aws-sdk/client-ec2");
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create anAmazon EC2 service client object.
const ec2Client = new EC2Client({ region: REGION });
module.exports = { ec2Client };
```

This example code can be found here on GitHub.

Create a Node.js module with the file name `ec2_describekeypairs.js`. Be sure to configure the SDK as previously shown. Create an empty JSON object to hold the parameters needed by the `DescribeKeyPairsCommand` method to return descriptions for all your key pairs. You can also provide an array of names of key pairs in the `KeyName` portion of the parameters in the JSON file to the `DescribeKeyPairsCommand` method.

```
// Import required AWS SDK clients and commands for Node.js
import { DescribeKeyPairsCommand } from "@aws-sdk/client-ec2";
import { ec2Client } from "./libs/ec2Client";
const run = async () => {
  try {
    const data = await ec2Client.send(new DescribeKeyPairsCommand({}));
    console.log("Success", JSON.stringify(data.KeyPairs));
    return data;
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

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

```
node ec2_describekeypairs.js
```

This example code can be found here on GitHub.

Creating a key pair

Create a `libs` directory, and create a Node.js module with the file name `ec2Client.js`. Copy and paste the code below into it, which creates the Amazon EC2 client object. Replace *REGION* with your AWS Region.

```
const  { EC2Client } = require( "@aws-sdk/client-ec2");
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create anAmazon EC2 service client object.
const ec2Client = new EC2Client({ region: REGION });
module.exports = { ec2Client };
```

This example code can be found here on GitHub.

Each key pair requires a name. Amazon EC2 associates the public key with the name that you specify as the key name. Create a Node.js module with the file name `ec2_createkeypair.js`. Be sure to configure the SDK as previously shown, including installing the required clients and packages. Create the JSON parameters to specify the name of the key pair, then pass them to call the `CreateKeyPairCommand` method.

> **Note**
> Replace *MY_KEY_PAIR* with the name of the key pair.

```
// Import required AWS SDK clients and commands for Node.js
import { CreateKeyPairCommand } from "@aws-sdk/client-ec2";
import { ec2Client } from "./libs/ec2Client";

// Set the parameters
const params = { KeyName: "MY_KEY_PAIR" }; //MY_KEY_PAIR

const run = async () => {
  try {
    const data = await ec2Client.send(new CreateKeyPairCommand(params));
    console.log(JSON.stringify(data));
    return data;
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

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

```
node ec2_createkeypair.js
```

The example code can be found here on GitHub.

Deleting a key pair

Create a `libs` directory, and create a Node.js module with the file name `ec2Client.js`. Copy and paste the code below into it, which creates the Amazon EC2 client object. Replace *REGION* with your AWS Region.

```
const  { EC2Client } = require( "@aws-sdk/client-ec2");
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
```

```
// Create anAmazon EC2 service client object.
const ec2Client = new EC2Client({ region: REGION });
module.exports = { ec2Client };
```

This example code can be found here on GitHub.

Create a Node.js module with the file name `ec2_deletekeypair.js`. Be sure to configure the SDK as previously shown, including installing the required clients and packages. To access Amazon EC2, create an `EC2` client service object. Create the JSON parameters to specify the name of the key pair you want to delete. Then call the `DeleteKeyPairCommand` method.

Note
Replace *`KEY_PAIR_NAME`* with the name of the key pair you want to delete.

```
// Import required AWS SDK clients and commands for Node.js
import { DeleteKeyPairCommand } from "@aws-sdk/client-ec2";
import { ec2Client } from "./libs/ec2Client";

// Set the parameters
const params = { KeyName: "KEY_PAIR_NAME" }; //KEY_PAIR_NAME

const run = async () => {
  try {
    const data = await ec2Client.send(new DeleteKeyPairCommand(params));
    console.log("Key Pair Deleted");
    return data;
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

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

```
node ec2_deletekeypair.js
```

This example code can be found here on GitHub.

Using Regions and Availability Zones with Amazon EC2

This Node.js code example shows:

- How to retrieve descriptions for AWS Regions and Availability Zones.

The scenario

Amazon EC2 is hosted in multiple locations worldwide. These locations are composed of Regions and Availability Zones. Each Region is a separate geographic area. Each Region has multiple, isolated locations known as *Availability Zones*. Amazon EC2 provides the ability to place instances and data in multiple locations.

In this example, you use a series of Node.js modules to retrieve details about Regions and Availability Zones. The Node.js modules use the SDK for JavaScript to manage instances by using the following methods of the Amazon EC2 client class:

- `DescribeAvailabilityZonesCommand`
- `DescribeRegionsCommand`

For more information about Regions and Availability Zones, see Regions and Availability Zones in the *Amazon EC2 User Guide for Linux Instances* or Regions and Availability Zones in the *Amazon EC2 User Guide for Windows Instances*.

Prerequisite tasks

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

- Create a shared configurations file with your user credentials. For more information about providing a shared credentials file, see Loading credentials in Node.js from the shared credentials file (p. 33).

> **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 sytax, please refer to JavaScript ES6/CommonJS syntax (p. 56)

Describing Regions and Availability Zones

Create a `libs` directory, and create a Node.js module with the file name `ec2Client.js`. Copy and paste the code below into it, which creates the Amazon EC2 client object. Replace ***`REGION`*** with your AWS Region.

```
const  { EC2Client } = require( "@aws-sdk/client-ec2");
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create anAmazon EC2 service client object.
const ec2Client = new EC2Client({ region: REGION });
module.exports = { ec2Client };
```

This example code can be found here on GitHub.

Create a Node.js module with the file name `ec2_describeregionsandzones.js`. Be sure to configure the SDK as previously shown. Create an empty JSON object to pass as parameters, which returns all available descriptions. Then call the `DescribeRegionsCommand` and `DescribeAvailabilityZonesCommand` methods.

```
// Import required AWS SDK clients and commands for Node.js
import { DescribeRegionsCommand } from "@aws-sdk/client-ec2";
import { ec2Client } from "./libs/ec2Client";

const run = async () => {
  try {
    const data = await ec2Client.send(new DescribeRegionsCommand({}));
    console.log("Availability Zones: ", data.Regions);
    return data;
  } catch (err) {
    console.log("Error", err);
  }
};
```

```
run();
```

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

```
node ec2_describeregionsandzones.js
```

This example code can be found here on GitHub.

Working with security groups in Amazon EC2

This Node.js code example shows:

- How to retrieve information about your security groups.
- How to create a security group to access an Amazon EC2 instance.
- How to delete an existing security group.

The scenario

An Amazon EC2 security group acts as a virtual firewall that controls the traffic for one or more instances. You add rules to each security group to allow traffic to or from its associated instances. You can modify the rules for a security group at any time; the new rules are automatically applied to all instances that are associated with the security group.

In this example, you use a series of Node.js modules to perform several Amazon EC2 operations involving security groups. The Node.js modules use the SDK for JavaScript to manage instances by using the following methods of the Amazon EC2 client class:

- `DescribeSecurityGroupsCommand`
- `AuthorizeSecurityGroupIngressCommand`
- `CreateSecurityGroupCommand`
- `DescribeVpcsCommand`
- `DeleteSecurityGroupCommand`

For more information about the Amazon EC2 security groups, see Amazon EC2 Amazon securitiy groups for Linux nstances in the *Amazon EC2 User Guide for Linux Instances* or Amazon EC2 Security groups for Windows instances in the *Amazon EC2 User Guide for Windows Instances*.

Prerequisite tasks

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

- Create a shared configurations file with your user credentials. For more information about providing a shared credentials file, see Loading credentials in Node.js from the shared credentials file (p. 33).

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 sytax, please refer to JavaScript ES6/CommonJS syntax (p. 56)

Describing your security groups

Create a `libs` directory, and create a Node.js module with the file name `ec2Client.js`. Copy and paste the code below into it, which creates the Amazon EC2 client object. Replace *REGION* with your AWS Region.

```
const  { EC2Client } = require( "@aws-sdk/client-ec2");
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create anAmazon EC2 service client object.
const ec2Client = new EC2Client({ region: REGION });
module.exports = { ec2Client };
```

This example code can be found here on GitHub.

Create a Node.js module with the file name `ec2_describesecuritygroups.js`. Be sure to configure the SDK as previously shown. Create a JSON object to pass as parameters, including the group IDs for the security groups you want to describe. Then call the `DescribeSecurityGroupsCommand` method of the Amazon EC2 service object.

Note
Replace *SECURITY_GROUP_ID* with the group IDs for the security groups you want to describe.

```

// Import required AWS SDK clients and commands for Node.js
import { DescribeSecurityGroupsCommand } from "@aws-sdk/client-ec2";
import { ec2Client } from "./libs/ec2Client";

// Set the parameters
const params = { GroupIds: ["SECURITY_GROUP_ID"] }; //SECURITY_GROUP_ID

const run = async () => {
  try {
    const data = await ec2Client.send(
      new DescribeSecurityGroupsCommand(params)
    );
    console.log("Success", JSON.stringify(data.SecurityGroups));
    return data;
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

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

```
node ec2_describesecuritygroups.js
```

This example code can be found here on GitHub.

Creating a security group and rules

Create a `libs` directory, and create a Node.js module with the file name `ec2Client.js`. Copy and paste the code below into it, which creates the Amazon EC2 client object. Replace *REGION* with your AWS Region.

```
const  { EC2Client } = require( "@aws-sdk/client-ec2");
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create anAmazon EC2 service client object.
const ec2Client = new EC2Client({ region: REGION });
module.exports = { ec2Client };
```

This example code can be found here on GitHub.

Create a Node.js module with the file name `ec2_createsecuritygroup.js`. Be sure to configure the SDK as previously shown. Create a JSON object for the parameters that specify the name of the security group, a description, and the ID for the VPC. Pass the parameters to the `CreateSecurityGroupCommand` method.

After you successfully create the security group, you can define rules for allowing inbound traffic. Create a JSON object for parameters that specify the IP protocol and inbound ports on which the Amazon EC2 instance will receive traffic. Pass the parameters to the `AuthorizeSecurityGroupIngressCommand` method.

Note

Replace *`KEY_PAIR_NAME`* with the name of the key pair, *`DESCRIPTION`* with a description of the security group, *`SECURITY_GROUP_NAME`* with the name of the security group, and *`SECURITY_GROUP_ID`* with the ID of the security group.

```
// Import required AWS SDK clients and commands for Node.js
import {
  DescribeVpcsCommand,
  CreateSecurityGroupCommand,
  AuthorizeSecurityGroupIngressCommand,
} from "@aws-sdk/client-ec2";
import { ec2Client } from "./libs/ec2Client";

// Set the parameters
const params = { KeyName: "KEY_PAIR_NAME" }; //KEY_PAIR_NAME

// Variable to hold a ID of a VPC
const vpc = null;

const run = async () => {
  try {
    const data = await ec2Client.send(new DescribeVpcsCommand(params));
    return data;
    const vpc = data.Vpcs[0].VpcId;
    const paramsSecurityGroup = {
      Description: "DESCRIPTION", //DESCRIPTION
      GroupName: "SECURITY_GROUP_NAME", // SECURITY_GROUP_NAME
      VpcId: vpc,
    };
  } catch (err) {
    console.log("Error", err);
  }
  try {
    const data = await ec2Client.send(new CreateSecurityGroupCommand(params));
    const SecurityGroupId = data.GroupId;
    console.log("Success", SecurityGroupId);
    return data;
  } catch (err) {
    console.log("Error", err);
  }
  try {
    const paramsIngress = {
      GroupId: "SECURITY_GROUP_ID", //SECURITY_GROUP_ID
      IpPermissions: [
```

```
        {
          IpProtocol: "tcp",
          FromPort: 80,
          ToPort: 80,
          IpRanges: [{ CidrIp: "0.0.0.0/0" }],
        },
        {
          IpProtocol: "tcp",
          FromPort: 22,
          ToPort: 22,
          IpRanges: [{ CidrIp: "0.0.0.0/0" }],
        },
      ],
    };
    const data = await ec2Client.send(
      new AuthorizeSecurityGroupIngressCommand(paramsIngress)
    );
    console.log("Ingress Successfully Set", data);
    return data;
  } catch (err) {
    console.log("Cannot retrieve a VPC", err);
  }
};
run();
```

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

```
node ec2_createsecuritygroup.js
```

This example code can be found here on GitHub.

Deleting a security group

Create a Node.js module with the file name `ec2_deletesecuritygroup.js`. Be sure to configure the SDK as previously shown. To access Amazon EC2, create an `EC2` client service object. Create the JSON parameters to specify the name of the security group to delete. Then call the `DeleteSecurityGroupCommand` method.

> **Note**
> Replace *`SECURITY_GROUP_ID`* with the security group ID.

```
// Import required AWS SDK clients and commands for Node.js
import { DeleteSecurityGroupCommand } from "@aws-sdk/client-ec2";
import { ec2Client } from "./libs/ec2Client";
// Set the parameters
const params = { GroupId: "SECURITY_GROUP_ID" }; //SECURITY_GROUP_ID

const run = async () => {
  try {
    const data = await ec2Client.send(new DeleteSecurityGroupCommand(params));
    console.log("Security Group Deleted");
    return data;
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

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

```
node ec2_deletesecuritygroup.js
```

This example code can be found here on GitHub.

Using elastic IP addresses in Amazon EC2

This Node.js code example shows:

- How to retrieve descriptions of your Elastic IP addresses.
- How to allocate and release an Elastic IP address.
- How to associate an Elastic IP address with an Amazon EC2 instance.

The scenario

An *Elastic IP address* is a static IP address designed for dynamic cloud computing. An Elastic IP address is associated with your AWS account. It is a public IP address, which is reachable from the internet. If your instance does not have a public IP address, you can associate an Elastic IP address with your instance to enable communication with the internet.

In this example, you use a series of Node.js modules to perform several Amazon EC2 operations involving Elastic IP addresses. The Node.js modules use the SDK for JavaScript to manage Elastic IP addresses by using these methods of the Amazon EC2 client class:

- `DescribeAddressesCommand`
- `AllocateAddressCommand`
- `AssociateAddressCommand`
- `ReleaseAddressCommand`

For more information about Elastic IP addresses in Amazon EC2, see Elastic IP Addresses in the *Amazon EC2 User Guide for Linux Instances* or Elastic IP Addresses in the *Amazon EC2 User Guide for Windows Instances*.

Prerequisite tasks

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

- Create a shared configurations file with your user credentials. For more information about providing a shared credentials file, see Loading credentials in Node.js from the shared credentials file (p. 33).
- Create an Amazon EC2 instance. For more information about creating Amazon EC2 instances, see Amazon EC2 Instances in the *Amazon EC2 User Guide for Linux Instances* or Amazon EC2 Instances in the *Amazon EC2 User Guide for Windows Instances*.

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 sytax, please refer to JavaScript ES6/CommonJS syntax (p. 56)

Describing elastic IP addresses

Create a Node.js module with the file name `ec2_describeaddresses.js`. Be sure to configure the SDK as previously shown. Create a JSON object to pass as parameters, filtering the addresses returned by those in your VPC. To retrieve descriptions of all your Elastic IP addresses, omit a filter from the parameters JSON. Then call the `DescribeAddressesCommand` method of the Amazon EC2 service object.

```
// Import required AWS SDK clients and commands for Node.js
import { EC2Client, DescribeAddressesCommand } from "@aws-sdk/client-ec2";
import { ec2Client } from "./libs/ec2Client";
// Set the parameters
const params = {
  Filters: [{ Name: "domain", Values: ["vpc"] }],
};

const run = async () => {
  try {
    const data = await ec2Client.send(new DescribeAddressesCommand(params));
    console.log(JSON.stringify(data.Addresses));
    return data;
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

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

```
node ec2_describeaddresses.js
```

This example code can be found here on GitHub.

Allocating and associating an elastic IP address with an Amazon EC2 instance

Create a Node.js module with the file name `ec2_allocateaddress.js`. Be sure to configure the SDK as previously shown. To access Amazon EC2, create an `EC2` client service object. Create a JSON object for the parameters used to allocate an Elastic IP address, which in this case specifies the `Domain` is a VPC. Call the `AllocateAddressCommand` method of the Amazon EC2 service object.

If the call succeeds, the `data` parameter to the callback function has an `AllocationId` property that identifies the allocated Elastic IP address.

Create a JSON object for the parameters used to associate an Elastic IP address to an Amazon EC2 instance, including the `AllocationId` from the newly allocated address and the `InstanceId` of the Amazon EC2 instance. Then call the `AssociateAddressesCommand` method of the Amazon EC2 service object.

> **Note**
> Replace *`INSTANCE_ID`* with the ID of the Amazon EC2 instance.

```
// Import required AWS SDK clients and commands for Node.js
import {
  AllocateAddressCommand,
  AssociateAddressCommand,
} from "@aws-sdk/client-ec2";
```

```
import { ec2Client } from "./libs/ec2Client";

// Set the parameters
const paramsAllocateAddress = { Domain: "vpc" };

const run = async () => {
  try {
    const data = await ec2Client.send(
      new AllocateAddressCommand(paramsAllocateAddress)
    );
    console.log("Address allocated:", data.AllocationId);
    return data;
    var paramsAssociateAddress = {
      AllocationId: data.AllocationId,
      InstanceId: "INSTANCE_ID", //INSTANCE_ID
    };
  } catch (err) {
    console.log("Address Not Allocated", err);
  }
  try {
    const results = await ec2Client.send(
      new AssociateAddressCommand(paramsAssociateAddress)
    );
    console.log("Address associated:", results.AssociationId);
    return results;
  } catch (err) {
    console.log("Address Not Associated", err);
  }
};
run();
```

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

```
node ec2_allocateaddress.js
```

This example code can be found here on GitHub.

Releasing an elastic IP address

Create a Node.js module with the file name `ec2_releaseaddress.js`. Be sure to configure the SDK as previously shown, including installing the required clients and packages. To access Amazon EC2, create an `EC2` client service object. Create a JSON object for the parameters used to release an Elastic IP address, which in this case specifies the `AllocationId` for the Elastic IP address. Releasing an Elastic IP address also disassociates it from any Amazon EC2 instance. Call the `ReleaseAddressCommand` method of the Amazon EC2 service object.

> **Note**
> Replace *`ALLOCATION_ID`* with the alloaction ID for the Elastic IP address.

```
// Import required AWS SDK clients and commands for Node.js
import { ReleaseAddressCommand } from "@aws-sdk/client-ec2";
import { ec2Client } from "./libs/ec2Client";

// Set the parameters
const paramsReleaseAddress = { AllocationId: "ALLOCATION_ID" }; //ALLOCATION_ID

const run = async () => {
  try {
    const data = await ec2Client.send(new ReleaseAddressCommand({}));
    console.log("Address released");
    return data;
  } catch (err) {
```

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

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

```
node ec2_releaseaddress.js
```

This example code can be found here on GitHub.

AWS Elemental MediaConvert examples

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

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

Topics

Getting your account-specific endpoint for MediaConvert

This Node.js code example shows:

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

The scenario

In this example, you use a Node.js module to call MediaConvert and retrieve your account-specific endpoint. You can retrieve your endpoint URL from the service default endpoint and so do not yet need your account-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 Loading credentials in Node.js from the shared credentials file (p. 33).
- 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 sytax, please refer to JavaScript ES6/CommonJS syntax (p. 56)

Getting your endpoint URL

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

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

This example code can be found here on GitHub.

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

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

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

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

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

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

```
node emc_getendpoint.js
```

This example code can be found here on GitHub.

Creating and managing transcoding jobs in MediaConvert

This Node.js code example shows:

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

The scenario

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

- `CreateJobCommand`
- `CancelJobCommand`
- `GetJobCommand`
- `ListJobsCommand`

Prerequisite tasks

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

- Set up the project environment to run these Node TypeScript examples, and install the required AWS SDK for JavaScript and third-party modules. Follow the instructions on GitHub.
- Create a shared configurations file with your user credentials. For more information about providing a shared credentials file, see Loading credentials in Node.js from the shared credentials file (p. 33).
- 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 sytax, please refer to JavaScript ES6/CommonJS syntax (p. 56)

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 account-specific endpoint. To do this, set the `endpoint` parameter on `mediaconvert(endpoint)`.

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

Defining a simple transcoding job

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

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

This example code can be found here on GitHub.

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

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

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

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

```
          Type: "FILE_GROUP_SETTINGS",
          FileGroupSettings: {
            Destination: "OUTPUT_BUCKET_NAME", //OUTPUT_BUCKET_NAME, e.g., "s3://
BUCKET_NAME/"
          },
        },
        Outputs: [
          {
            VideoDescription: {
              ScalingBehavior: "DEFAULT",
              TimecodeInsertion: "DISABLED",
              AntiAlias: "ENABLED",
              Sharpness: 50,
              CodecSettings: {
                Codec: "H_264",
                H264Settings: {
                  InterlaceMode: "PROGRESSIVE",
                  NumberReferenceFrames: 3,
                  Syntax: "DEFAULT",
                  Softness: 0,
                  GopClosedCadence: 1,
                  GopSize: 90,
                  Slices: 1,
                  GopBReference: "DISABLED",
                  SlowPal: "DISABLED",
                  SpatialAdaptiveQuantization: "ENABLED",
                  TemporalAdaptiveQuantization: "ENABLED",
                  FlickerAdaptiveQuantization: "DISABLED",
                  EntropyEncoding: "CABAC",
                  Bitrate: 5000000,
                  FramerateControl: "SPECIFIED",
                  RateControlMode: "CBR",
                  CodecProfile: "MAIN",
                  Telecine: "NONE",
                  MinIInterval: 0,
                  AdaptiveQuantization: "HIGH",
                  CodecLevel: "AUTO",
                  FieldEncoding: "PAFF",
                  SceneChangeDetect: "ENABLED",
                  QualityTuningLevel: "SINGLE_PASS",
                  FramerateConversionAlgorithm: "DUPLICATE_DROP",
                  UnregisteredSeiTimecode: "DISABLED",
                  GopSizeUnits: "FRAMES",
                  ParControl: "SPECIFIED",
                  NumberBFramesBetweenReferenceFrames: 2,
                  RepeatPps: "DISABLED",
                  FramerateNumerator: 30,
                  FramerateDenominator: 1,
                  ParNumerator: 1,
                  ParDenominator: 1,
                },
              },
              AfdSignaling: "NONE",
              DropFrameTimecode: "ENABLED",
              RespondToAfd: "NONE",
              ColorMetadata: "INSERT",
            },
            AudioDescriptions: [
              {
                AudioTypeControl: "FOLLOW_INPUT",
                CodecSettings: {
                  Codec: "AAC",
                  AacSettings: {
                    AudioDescriptionBroadcasterMix: "NORMAL",
                    RateControlMode: "CBR",
                    CodecProfile: "LC",
```

```
                    CodingMode: "CODING_MODE_2_0",
                    RawFormat: "NONE",
                    SampleRate: 48000,
                    Specification: "MPEG4",
                    Bitrate: 64000,
                  },
                },
                LanguageCodeControl: "FOLLOW_INPUT",
                AudioSourceName: "Audio Selector 1",
              },
            ],
            ContainerSettings: {
              Container: "MP4",
              Mp4Settings: {
                CslgAtom: "INCLUDE",
                FreeSpaceBox: "EXCLUDE",
                MoovPlacement: "PROGRESSIVE_DOWNLOAD",
              },
            },
            NameModifier: "_1",
          },
        ],
      },
    ],
    AdAvailOffset: 0,
    Inputs: [
      {
        AudioSelectors: {
          "Audio Selector 1": {
            Offset: 0,
            DefaultSelection: "NOT_DEFAULT",
            ProgramSelection: 1,
            SelectorType: "TRACK",
            Tracks: [1],
          },
        },
        VideoSelector: {
          ColorSpace: "FOLLOW",
        },
        FilterEnable: "AUTO",
        PsiControl: "USE_PSI",
        FilterStrength: 0,
        DeblockFilter: "DISABLED",
        DenoiseFilter: "DISABLED",
        TimecodeSource: "EMBEDDED",
        FileInput: "INPUT_BUCKET_AND_FILENAME", //INPUT_BUCKET_AND_FILENAME, e.g., "s3://
BUCKET_NAME/FILE_NAME"
      },
    ],
    TimecodeConfig: {
      Source: "EMBEDDED",
    },
  },
};
```

Creating a transcoding job

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

```
const run = async () => {
  try {
    const data = await emcClient.send(new CreateJobCommand(params));
```

```
    console.log("Job created!", data);
    return data;
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

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

```
node emc_createjob.js
```

This full example code can be found here on GitHub.

Canceling a transcoding job

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

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

This example code can be found here on GitHub.

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

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

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

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

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

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

```
node ec2_canceljob.js
```

This example code can be found here on GitHub.

Listing recent transcoding jobs

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

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

This example code can be found here on GitHub.

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

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

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

```

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

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

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

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

```
node emc_listjobs.js
```

This example code can be found here on GitHub.

Using job templates in MediaConvert

This Node.js code example shows:

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

The scenario

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

- `CreateJobTemplateCommand`
- `CreateJobCommand`
- `DeleteJobTemplateCommand`
- `ListJobTemplatesCommand`

Prerequisite tasks

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

- Set up the project environment to run these Node TypeScript examples, and install the required AWS SDK for JavaScript and third-party modules. Follow the instructions on GitHub.
- Create a shared configurations file with your user credentials. For more information about providing a shared credentials file, see Loading credentials in Node.js from the shared credentials file (p. 33).
- 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 sytax, please refer to JavaScript ES6/CommonJS syntax (p. 56)

Creating a job template

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

```
import { MediaConvertClient } from "@aws-sdk/client-mediaconvert";
// Set the AWS Region.
const REGION = "REGION";
// 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 (p. 124).

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

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

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

const params = {
  Category: "YouTube Jobs",
  Description: "Final production transcode",
  Name: "DemoTemplate",
  Queue: "JOB_QUEUE_ARN", //JOB_QUEUE_ARN
  Settings: {
    OutputGroups: [
      {
        Name: "File Group",
        OutputGroupSettings: {
          Type: "FILE_GROUP_SETTINGS",
          FileGroupSettings: {
            Destination: "BUCKET_NAME", // BUCKET_NAME e.g., "s3://BUCKET_NAME/"
          },
        },
        Outputs: [
          {
            VideoDescription: {
              ScalingBehavior: "DEFAULT",
              TimecodeInsertion: "DISABLED",
              AntiAlias: "ENABLED",
              Sharpness: 50,
              CodecSettings: {
```

```
          Codec: "H_264",
          H264Settings: {
            InterlaceMode: "PROGRESSIVE",
            NumberReferenceFrames: 3,
            Syntax: "DEFAULT",
            Softness: 0,
            GopClosedCadence: 1,
            GopSize: 90,
            Slices: 1,
            GopBReference: "DISABLED",
            SlowPal: "DISABLED",
            SpatialAdaptiveQuantization: "ENABLED",
            TemporalAdaptiveQuantization: "ENABLED",
            FlickerAdaptiveQuantization: "DISABLED",
            EntropyEncoding: "CABAC",
            Bitrate: 5000000,
            FramerateControl: "SPECIFIED",
            RateControlMode: "CBR",
            CodecProfile: "MAIN",
            Telecine: "NONE",
            MinIInterval: 0,
            AdaptiveQuantization: "HIGH",
            CodecLevel: "AUTO",
            FieldEncoding: "PAFF",
            SceneChangeDetect: "ENABLED",
            QualityTuningLevel: "SINGLE_PASS",
            FramerateConversionAlgorithm: "DUPLICATE_DROP",
            UnregisteredSeiTimecode: "DISABLED",
            GopSizeUnits: "FRAMES",
            ParControl: "SPECIFIED",
            NumberBFramesBetweenReferenceFrames: 2,
            RepeatPps: "DISABLED",
            FramerateNumerator: 30,
            FramerateDenominator: 1,
            ParNumerator: 1,
            ParDenominator: 1,
          },
        },
        AfdSignaling: "NONE",
        DropFrameTimecode: "ENABLED",
        RespondToAfd: "NONE",
        ColorMetadata: "INSERT",
      },
      AudioDescriptions: [
        {
          AudioTypeControl: "FOLLOW_INPUT",
          CodecSettings: {
            Codec: "AAC",
            AacSettings: {
              AudioDescriptionBroadcasterMix: "NORMAL",
              RateControlMode: "CBR",
              CodecProfile: "LC",
              CodingMode: "CODING_MODE_2_0",
              RawFormat: "NONE",
              SampleRate: 48000,
              Specification: "MPEG4",
              Bitrate: 64000,
            },
          },
          LanguageCodeControl: "FOLLOW_INPUT",
          AudioSourceName: "Audio Selector 1",
        },
      ],
      ContainerSettings: {
        Container: "MP4",
        Mp4Settings: {
```

```
              CslgAtom: "INCLUDE",
              FreeSpaceBox: "EXCLUDE",
              MoovPlacement: "PROGRESSIVE_DOWNLOAD",
            },
          },
          NameModifier: "_1",
        },
      ],
    },
  ],
  AdAvailOffset: 0,
  Inputs: [
    {
      AudioSelectors: {
        "Audio Selector 1": {
          Offset: 0,
          DefaultSelection: "NOT_DEFAULT",
          ProgramSelection: 1,
          SelectorType: "TRACK",
          Tracks: [1],
        },
      },
      VideoSelector: {
        ColorSpace: "FOLLOW",
      },
      FilterEnable: "AUTO",
      PsiControl: "USE_PSI",
      FilterStrength: 0,
      DeblockFilter: "DISABLED",
      DenoiseFilter: "DISABLED",
      TimecodeSource: "EMBEDDED",
    },
  ],
  TimecodeConfig: {
    Source: "EMBEDDED",
  },
 },
};

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

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

```
node emc_create_jobtemplate.js
```

This example code can be found here on GitHub.

Creating a transcoding job from a job template

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

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

This example code can be found here on GitHub.

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

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

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

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

const params = {
  Queue: "QUEUE_ARN", //QUEUE_ARN
  JobTemplate: "TEMPLATE_NAME", //TEMPLATE_NAME
  Role: "ROLE_ARN", //ROLE_ARN
  Settings: {
    Inputs: [
      {
        AudioSelectors: {
          "Audio Selector 1": {
            Offset: 0,
            DefaultSelection: "NOT_DEFAULT",
            ProgramSelection: 1,
            SelectorType: "TRACK",
            Tracks: [1],
          },
        },
        VideoSelector: {
          ColorSpace: "FOLLOW",
        },
        FilterEnable: "AUTO",
        PsiControl: "USE_PSI",
        FilterStrength: 0,
        DeblockFilter: "DISABLED",
        DenoiseFilter: "DISABLED",
        TimecodeSource: "EMBEDDED",
        FileInput: "INPUT_BUCKET_AND_FILENAME", //INPUT_BUCKET_AND_FILENAME, e.g., "s3://
BUCKET_NAME/FILE_NAME"
      },
    ],
  },
};

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

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

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

```
node emc_template_createjob.js
```

This example code can be found here on GitHub.

Listing your job templates

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

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

This example code can be found here on GitHub.

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

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

```

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

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

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

```
  }
};
run();
```

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

```
node emc_listtemplates.js
```

This example code can be found here on GitHub.

Deleting a job template

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

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

This example code can be found here on GitHub.

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

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

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

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

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

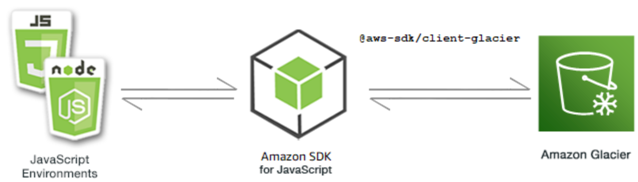
To run the example, enter the following at the command prompt.

```
node emc_deletetemplate.js
```

This example code can be found here on GitHub.

Amazon S3 Glacier examples

Amazon S3 Glacier is a secure cloud storage service for data archiving and long-term backup. The service is optimized for infrequently accessed data where a retrieval time of several hours is suitable.



The JavaScript API for Amazon S3 Glacier is exposed through the `Glacier` client class. For more information about using the S3 Glacier client class, see Class: Glacier in the API reference.

Topics

Creating a S3 Glacier vault

This Node.js code example shows:

- How to create a vault using the `CreateVaultCommand` method of the Amazon S3 Glacier service object.

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 Loading credentials in Node.js from the shared credentials file (p. 33).

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 (p. 56).

Create the vault

Create a `libs` directory, and create a Node.js module with the file name `glacierClient.js`. Copy and paste the code below into it, which creates the S3 Glacier client object. Replace *REGION* with your AWS Region.

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

This example code can be found here on GitHub.

Create a Node.js module with the file name `createVault.js`. Copy and paste the code below into it.

Note
Replace *VAULT_NAME* with the name of the S3 Glacier 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();
```

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

```
node createVault.js
```

This example code can be found here on GitHub.

Uploading an archive to S3 Glacier

This Node.js code example shows:

- How to upload an archive to Amazon S3 Glacier using the `uploadArchive` method of the S3 Glacier service object.

The following example uploads a single `Buffer` object as an entire archive using the `UploadArchiveCommand` method of the S3 Glacier service object.

The example assumes you've already created a vault named `VAULT_NAME`. The SDK automatically computes the tree hash `checksum` for the data uploaded, however, you can override it by passing your own checksum parameter.

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 Loading credentials in Node.js from the shared credentials file (p. 33).

> **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 (p. 56).

Upload the archive

Create a `libs` directory, and create a Node.js module with the file name `glacierClient.js`. Copy and paste the code below into it, which creates the S3 Glacier client object. Replace ***`REGION`*** with your AWS Region.

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

This example code can be found here on GitHub.

Create a Node.js module with the file name `uploadArchive.js`. Copy and paste the code below into it.

> **Note**
> Replace ***`VAULT_NAME`*** with the name of the S3 Glacier vault.

```
// 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();
```

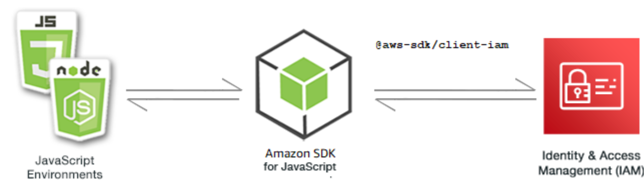
To run the example, enter the following at the command prompt.

```
node uploadArchive.js
```

This example code can be found here on GitHub.

AWS Identity and Access Management examples

AWS Identity and Access Management (IAM) is a web service that enables Amazon Web Services (AWS) customers to manage users and user permissions in AWS. The service is targeted at organizations with multiple users or systems in the cloud that use AWS products. With IAM, you can centrally manage users, security credentials such as access keys, and permissions that control which AWS resources users can access.



The JavaScript API for IAM is exposed through the `IAM` client class. For more information about using the IAM client class, see Class: IAM in the API Reference.

Topics

Managing IAM users

This Node.js code example shows:

- How to retrieve a list of IAM users.

- How to create and delete users.
- How to update a user name.

The scenario

In this example, a series of Node.js modules are used to create and manage users in IAM. The Node.js modules use the SDK for JavaScript to create, delete, and update users using these methods of the `IAM` client class:

- `CreateUserCommand`
- `ListUsersCommand`
- `UpdateUserCommand`
- `GetUserCommand`
- `DeleteUserCommand`

For more information about IAM users, see IAM users in the *IAM User 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 Loading credentials in Node.js from the shared credentials file (p. 33).

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 (p. 56).

Creating a user

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

```
import { IAMClient } from "@aws-sdk/client-iam";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an IAM service client object.
const iamClient = new IAMClient({ region: REGION });
export { iamClient };
```

This example code can be found here on GitHub.

Create a Node.js module with the file name `iam_createuser.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. Create a JSON object containing the parameters needed, which consists of the user name you want to use for the new user as a command-line parameter.

Call the `GetUserCommand` method of the `IAM` client service object to see if the user name already exists. If the user name does not currently exist, call the `CreateUserCommand` method to create it. If the name already exists, write a message to that effect to the console.

> **Note**
> Replace *USER_NAME* with the user name to create.

```
// Import required AWS SDK clients and commands for Node.js
import { iamClient } from "./libs/iamClient.js";
import { GetUserCommand, CreateUserCommand } from "@aws-sdk/client-iam";

// Set the parameters
const params = { UserName: "USER_NAME" }; //USER_NAME

const run = async () => {
  try {
    const data = await iamClient.send(new GetUserCommand(params));
    console.log(
      "User " + process.argv[3] + " already exists",
      data.User.UserId
    );
    return data;
  } catch (err) {
    try {
      const results = await iamClient.send(new CreateUserCommand(params));
      console.log("Success", results);
      return results;
    } catch (err) {
      console.log("Error", err);
    }
  }
};
run();
```

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

```
node iam_createuser.js
```

This example code can be found here on GitHub.

Listing users in Your Account

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

```
import { IAMClient } from "@aws-sdk/client-iam";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an IAM service client object.
const iamClient = new IAMClient({ region: REGION });
export { iamClient };
```

This example code can be found here on GitHub.

Create a Node.js module with the file name `iam_listusers.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. Create a JSON object containing the parameters needed to list your users, limiting the number returned by setting the `MaxItems` parameter to 10. Call the `ListUsersCommand` method of the `IAM` client service object. Write the first user's name and creation date to the console.

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

// Set the parameters
const params = { MaxItems: 10 };

const run = async () => {
  try {
    const data = await iamClient.send(new ListUsersCommand(params));
    return data;
    const users = data.Users || [];
    users.forEach(function (user) {
      console.log("User " + user.UserName + " created", user.CreateDate);
    });
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

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

```
node iam_listusers.js
```

This example code can be found here on GitHub.

Updating a user's name

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

```
import { IAMClient } from "@aws-sdk/client-iam";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an IAM service client object.
const iamClient = new IAMClient({ region: REGION });
export { iamClient };
```

This example code can be found here on GitHub.

Create a Node.js module with the file name `iam_updateuser.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. Create a JSON object containing the parameters needed to list your users, specifying both the current and new user names as command-line parameters. Call the `UpdateUserCommand` method of the `IAM` client service object.

> **Note**
> Replace *ORIGNAL_USER_NAME* with the user name to update, and *NEW_USER_NAME* with the new user name.

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

// Set the parameters
const params = {
  UserName: "ORIGINAL_USER_NAME", //ORIGINAL_USER_NAME
  NewUserName: "NEW_USER_NAME", //NEW_USER_NAME
};
```

```
const run = async () => {
  try {
    const data = await iamClient.send(new UpdateUserCommand(params));
    console.log("Success, username updated");
    return data;
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

To run the example, enter the following at the command prompt, specifying the user's current name followed by the new user name.

```
node iam_updateuser.js
```

This example code can be found here on GitHub.

Deleting a user

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

```
import { IAMClient } from "@aws-sdk/client-iam";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an IAM service client object.
const iamClient = new IAMClient({ region: REGION });
export { iamClient };
```

This example code can be found here on GitHub.

Create a Node.js module with the file name `iam_deleteuser.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. Create a JSON object containing the parameters needed, which consists of the user name to delete as a command-line parameter.

Call the `GetUserCommand` method of the `IAM` client service object to see if the user name already exists. If the user name does not currently exist, write a message to that effect to the console. If the user exists, call the `DeleteUserCommand` method to delete it.

> **Note**
> Replace *USER_NAME* with the name of the user to delete.

```
// Import required AWS SDK clients and commands for Node.js
import { iamClient } from "./libs/iamClient.js";
import { DeleteUserCommand, GetUserCommand } from "@aws-sdk/client-iam";

// Set the parameters
const params = { UserName: "USER_NAME" }; //USER_NAME

const run = async () => {
  try {
    const data = await iamClient.send(new GetUserCommand(params));
    return data;
    try {
      const results = await iamClient.send(new DeleteUserCommand(params));
      console.log("Success", results);
      return results;
```

```
    } catch (err) {
      console.log("Error", err);
    }
  } catch (err) {
    console.log("User " + process.argv[2] + " does not exist.");
  }
};
run();
```

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

```
node iam_deleteuser.js
```

This example code can be found here on GitHub.

Working with IAM policies

This Node.js code example shows:

- How to create and delete IAM policies.
- How to attach and detach IAM policies from roles.

The scenario

You grant permissions to a user by creating a *policy*, which is a document that lists the actions that a user can perform and the resources those actions can affect. Any actions or resources that are not explicitly allowed are denied by default. Policies can be created and attached to users, groups of users, roles assumed by users, and resources.

In this example, a series of Node.js modules are used to manage policies in IAM. The Node.js modules use the SDK for JavaScript to create and delete policies as well as attaching and detaching role policies using these methods of the `IAM` client class:

- `CreatePolicyCommand`
- `GetPolicyCommand`
- `ListAttachedRolePoliciesCommand`
- `AttachRolePolicyCommand`
- `DetachRolePolicyCommand`

For more information about IAM users, see Overview of access management: Permissions and policies in the *IAM User 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 Loading credentials in Node.js from the shared credentials file (p. 33).
- Create an IAM role to which you can attach policies. For more information about creating roles, see Creating IAM roles in the *IAM User 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 (p. 56).

Creating an IAM policy

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

```
import { IAMClient } from "@aws-sdk/client-iam";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an IAM service client object.
const iamClient = new IAMClient({ region: REGION });
export { iamClient };
```

This example code can be found here on GitHub.

Create a Node.js module with the file name `iam_createpolicy.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. Create two JSON objects, one containing the policy document to create and the other containing the parameters needed to create the policy, which includes the policy JSON and the name to give the policy. Be sure to stringify the policy JSON object in the parameters. Call the `CreatePolicyCommand` method of the `IAM` client service object.

Note
Replace *`RESOURCE_ARN`* with the Amazon Resource Name (ARN) of the resource you want to grant the permissions to, and *`DYNAMODB_POLICY_NAME`* with the name of the DynamoDB policy name.

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

// Set the parameters
const myManagedPolicy = {
  Version: "2012-10-17",
  Statement: [
    {
      Effect: "Allow",
      Action: "logs:CreateLogGroup",
      Resource: "RESOURCE_ARN", // RESOURCE_ARN
    },
    {
      Effect: "Allow",
      Action: [
        "dynamodb:DeleteItem",
        "dynamodb:GetItem",
        "dynamodb:PutItem",
```

```
        "dynamodb:Scan",
        "dynamodb:UpdateItem",
      ],
      Resource: "DYNAMODB_POLICY_NAME", // DYNAMODB_POLICY_NAME; e.g., "myDynamoDBName"
    },
  ],
};
const params = {
  PolicyDocument: JSON.stringify(myManagedPolicy),
  PolicyName: process.argv[4],
};

const run = async () => {
  try {
    const data = await iamClient.send(new CreatePolicyCommand(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 iam_createpolicy.js
```

This example code can be found here on GitHub.

Getting an IAM policy

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

```
import { IAMClient } from "@aws-sdk/client-iam";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an IAM service client object.
const iamClient = new IAMClient({ region: REGION });
export { iamClient };
```

This example code can be found here on GitHub.

Create a Node.js module with the file name `iam_getpolicy.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. Create a JSON object containing the parameters needed retrieve a policy, which is the ARN of the policy to get. Call the `GetPolicyCommand` method of the `IAM` client service object. Write the policy description to the console.

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

// Set the parameters
const params = {
  PolicyArn: "arn:aws:iam::aws:policy/AWSLambdaExecute",
};

const run = async () => {
  try {
    const data = await iamClient.send(new GetPolicyCommand(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 iam_getpolicy.js
```

This example code can be found here on GitHub.

Attaching a managed role policy

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

```
import { IAMClient } from "@aws-sdk/client-iam";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an IAM service client object.
const iamClient = new IAMClient({ region: REGION });
export { iamClient };
```

This example code can be found here on GitHub.

Create a Node.js module with the file name `iam_attachrolepolicy.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. Create a JSON object containing the parameters needed to get a list of managed IAM policies attached to a role, which consists of the name of the role. Provide the role name as a command-line parameter. Call the `ListAttachedRolePoliciesCommand` method of the `IAM` client service object, which returns an array of managed policies to the callback function.

Check the array members to see if the policy to attach to the role is already attached. If the policy is not attached, call the `AttachRolePolicyCommand` method to attach it.

> **Note**
> Replace *ROLE_NAME* with the name of the role to attach.

```
// Import required AWS SDK clients and commands for Node.js
import { iamClient } from "./libs/iamClient.js";
import {
  ListAttachedRolePoliciesCommand,
  AttachRolePolicyCommand,
} from "@aws-sdk/client-iam";

// Set the parameters
const ROLENAME = "ROLE_NAME";
const paramsRoleList = { RoleName: ROLENAME }; //ROLE_NAME
const params = {
  PolicyArn: "arn:aws:iam::aws:policy/AmazonDynamoDBFullAccess",
  RoleName: ROLENAME,
};
const run = async () => {
  try {
    const data = await iamClient.send(
      new ListAttachedRolePoliciesCommand(paramsRoleList)
    );
    return data;
```

```
    const 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();
      }
    });
    try {
      const data = await iamClient.send(new AttachRolePolicyCommand(params));
      console.log("Role attached successfully");
      return data;
    } catch (err) {
      console.log("Error", err);
    }
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

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

```
node iam_attachrolepolicy.js
```

This example code can be found here on GitHub.

Detaching a managed role policy

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

```
import { IAMClient } from "@aws-sdk/client-iam";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an IAM service client object.
const iamClient = new IAMClient({ region: REGION });
export { iamClient };
```

This example code can be found here on GitHub.

Create a Node.js module with the file name `iam_detachrolepolicy.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. Create a JSON object containing the parameters needed to get a list of managed IAM policies attached to a role, which consists of the name of the role. Provide the role name as a command-line parameter. Call the `ListAttachedRolePoliciesCommand` method of the `IAM` client service object, which returns an array of managed policies in the callback function.

Check the array members to see if the policy to detach from the role is attached. If the policy is attached, call the `DetachRolePolicyCommand` method to detach it.

Note
Replace *ROLE_NAME* with the name of the role to detach.

```
// Import required AWS SDK clients and commands for Node.js
import { iamClient } from "./libs/iamClient.js";
import {
  ListAttachedRolePoliciesCommand,
  DetachRolePolicyCommand,
} from "@aws-sdk/client-iam";
```

```
// Set the parameters
const params = { RoleName: "ROLE_NAME" }; //ROLE_NAME

const run = async () => {
  try {
    const data = await iamClient.send(
      new ListAttachedRolePoliciesCommand(params)
    );
    return data;
    const myRolePolicies = data.AttachedPolicies;
    myRolePolicies.forEach(function (val, index, array) {
      if (myRolePolicies[index].PolicyName === "AmazonDynamoDBFullAccess") {
        const params = {
          PolicyArn: "arn:aws:iam::aws:policy/AmazonDynamoDBFullAccess",
          paramsRoleList,
        };
        try {
          const results = iamClient.send(
            new DetachRolePolicyCommand(paramsRoleList)
          );
          console.log("Policy detached from role successfully");
          process.exit();
        } catch (err) {
          console.log("Unable to detach policy from role", err);
        }
      } else {
      }
    });
  } catch (err) {
    console.log("User " + process.argv[2] + " does not exist.");
  }
};
run();
```

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

```
node iam_detachrolepolicy.js
```

This example code can be found here on GitHub.

Managing IAM access keys

This Node.js code example shows:

- How to manage the access keys of your users.

The scenario

Users need their own access keys to make programmatic calls to AWS from the SDK for JavaScript. To fill this need, you can create, modify, view, or rotate access keys (access key IDs and secret access keys) for IAM users. By default, when you create an access key, its status is `Active`, which means the user can use the access key for API calls.

In this example, a series of Node.js modules are used manage access keys in IAM. The Node.js modules use the SDK for JavaScript to manage IAM access keys using these methods of the `IAM` client class:

- CreateAccessKeyCommand
- ListAccessKeysCommand
- GetAccessKeyLastUsedCommand
- UpdateAccessKeyCommand
- DeleteAccessKeyCommand

For more information about IAM access keys, see Access keys in the *IAM User 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 Loading credentials in Node.js from the shared credentials file (p. 33).

> **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 (p. 56).

Creating access keys for a user

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

```
import { IAMClient } from "@aws-sdk/client-iam";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an IAM service client object.
const iamClient = new IAMClient({ region: REGION });
export { iamClient };
```

This example code can be found here on GitHub.

Create a Node.js module with the file name `iam_createaccesskeys.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. Create a JSON object containing the parameters needed to create new access keys, which includes IAM user's name. Call the `CreateAccessKeyCommand` method of the `IAM` client service object.

> **Note**
> Replace ***`IAM_USER_NAME`*** with the IAM user name.

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

// Set the parameters
```

```
const params = {UserName: "IAM_USER_NAME"}; //IAM_USER_NAME

const run = async () => {
  try {
    const data = await iamClient.send(new CreateAccessKeyCommand(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. Be sure to pipe the returned data to a text file in order not to lose the secret key, which can only be provided once.

```
node iam_createaccesskeys.js > newuserkeysV3.txt
```

This example code can be found here on GitHub.

Listing a user's access keys

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

```
import { IAMClient } from "@aws-sdk/client-iam";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an IAM service client object.
const iamClient = new IAMClient({ region: REGION });
export { iamClient };
```

This example code can be found here on GitHub.

Create a Node.js module with the file name `iam_listaccesskeys.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. Create a JSON object containing the parameters needed to retrieve the user's access keys, which includes IAM user's name and optionally the maximum number of access key pairs listed. Call the `ListAccessKeysCommand` method of the `IAM` client service object.

Note

Replace *IAM_USER_NAME* with the IAM user name.

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

// Set the parameters
const params = {
  MaxItems: 5,
  UserName: "IAM_USER_NAME", //IAM_USER_NAME
};

const run = async () => {
  try {
    const data = await iamClient.send(new ListAccessKeysCommand(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 iam_listaccesskeys.js
```

This example code can be found here on GitHub.

Getting the last use for access keys

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

```
import { IAMClient } from "@aws-sdk/client-iam";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an IAM service client object.
const iamClient = new IAMClient({ region: REGION });
export { iamClient };
```

This example code can be found here on GitHub.

Create a Node.js module with the file name `iam_accesskeylastused.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. Create a JSON object containing the parameters needed to create new access keys, which is the access key ID for which the last use information. Call the `GetAccessKeyLastUsedCommand` method of the `IAM` service object.

> **Note**
> Replace *ACCESS_KEY_ID* with the access key ID for which the the last use information.

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

// Set the parameters
const params = { AccessKeyId: "ACCESS_KEY_ID" }; //ACCESS_KEY_ID

const run = async () => {
  try {
    const data = await iamClient.send(new GetAccessKeyLastUsedCommand(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 iam_accesskeylastused.js
```

This example code can be found here on GitHub.

Updating access key status

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

```
import { IAMClient } from "@aws-sdk/client-iam";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an IAM service client object.
const iamClient = new IAMClient({ region: REGION });
export { iamClient };
```

This example code can be found here on GitHub.

Create a Node.js module with the file name `iam_updateaccesskey.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. Create a JSON object containing the parameters needed to update the status of an access keys, which includes the access key ID and the updated status. The status can be `Active` or `Inactive`. Call the `updateAccessKey` method of the `IAM` client service object.

Note

Replace *`ACCESS_KEY_ID`* the access key ID and the updated status, and *`USER_NAME`* with the name of the user.

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

// Set the parameters
const params = {
  AccessKeyId: "ACCESS_KEY_ID", //ACCESS_KEY_ID
  Status: "Active",
  UserName: "USER_NAME", //USER_NAME
};

const run = async () => {
  try {
    const data = await iamClient.send(new UpdateAccessKeyCommand(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 iam_updateaccesskey.js
```

This example code can be found here on GitHub.

Deleting access keys

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

```
import { IAMClient } from "@aws-sdk/client-iam";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an IAM service client object.
const iamClient = new IAMClient({ region: REGION });
export { iamClient };
```

This example code can be found here on GitHub.

Create a Node.js module with the file name `iam_deleteaccesskey.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. Create a JSON object containing the parameters needed to delete access keys, which includes the access key ID and the name of the user. Call the `DeleteAccessKeyCommand` method of the `IAM` client service object.

> **Note**
> Replace *ACCESS_KEY_ID* with your access key ID, and *USER_NAME* with the user name.

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

// Set the parameters
const params = {
  AccessKeyId: "ACCESS_KEY_ID", // ACCESS_KEY_ID
  UserName: "USER_NAME", // USER_NAME
};

const run = async () => {
  try {
    const data = await iamClient.send(new DeleteAccessKeyCommand(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 iam_deleteaccesskey.js
```

This example code can be found here on GitHub.

Working with IAM server certificates

This Node.js code example shows:

- How to carry out basic tasks in managing server certificates for HTTPS connections.

The scenario

To enable HTTPS connections to your website or application on AWS, you need an SSL/TLS *server certificate*. To use a certificate that you obtained from an external provider with your website or application onAWS, you must upload the certificate to IAM or import it into AWS Certificate Manager.

In this example, a series of Node.js modules are used to handle server certificates in IAM. The Node.js modules use the SDK for JavaScript to manage server certificates using these methods of the `IAM` client class:

- `ListServerCertificatesCommand`
- `GetServerCertificateCommand`
- `UpdateServerCertificateCommand`
- `DeleteServerCertificatecommand`

For more information about server certificates, see Working with server certificates in the *IAM User 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 Loading credentials in Node.js from the shared credentials file (p. 33).

> **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 (p. 56).

Listing your server certificates

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

```
import { IAMClient } from "@aws-sdk/client-iam";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an IAM service client object.
const iamClient = new IAMClient({ region: REGION });
export { iamClient };
```

This example code can be found here on GitHub.

Create a Node.js module with the file name `iam_listservercerts.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. Call the `ListServerCertificatesCommand` method of the `IAM` client service object.

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

const run = async () => {
  try {
    const data = await iamClient.send(new ListServerCertificatesCommand({}));
    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 iam_listservercers.js
```

This example code can be found here on GitHub.

Getting a server certificate

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

```
import { IAMClient } from "@aws-sdk/client-iam";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an IAM service client object.
const iamClient = new IAMClient({ region: REGION });
export { iamClient };
```

This example code can be found here on GitHub.

Create a Node.js module with the file name `iam_getservercert.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. Create a JSON object containing the parameters needed get a certificate, which consists of the name of the server certificate. Call the `GetServerCertificatesCommand` method of the `IAM` client service object.

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

// Set the parameters
const params = { ServerCertificateName: "CERTIFICATE_NAME" }; //CERTIFICATE_NAME

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

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

```
node iam_getservercert.js
```

This example code can be found here on GitHub.

Updating a server certificate

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

```
import { IAMClient } from "@aws-sdk/client-iam";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an IAM service client object.
const iamClient = new IAMClient({ region: REGION });
export { iamClient };
```

This example code can be found here on GitHub.

Create a Node.js module with the file name `iam_updateservercert.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. Create a JSON object containing the parameters needed to update a certificate, which consists of the name of the existing server certificate as well as the name of the new certificate. Call the `UpdateServerCertificateCommand` method of the `IAM` client service object.

> **Note**
> Replace *`CERTIFICATE_NAME`* with the service certicate name to update, and *`NEW_CERTIFICATE_NAME`* with the new certificate name.

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

// Set the parameters
const params = {
  ServerCertificateName: "CERTIFICATE_NAME", //CERTIFICATE_NAME
  NewServerCertificateName: "NEW_CERTIFICATE_NAME", //NEW_CERTIFICATE_NAME
};

const run = async () => {
  try {
    const data = await iamClient.send(
      new UpdateServerCertificateCommand(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 iam_updateservercert.js
```

This example code can be found here on GitHub.

Deleting a server certificate

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

```
import { IAMClient } from "@aws-sdk/client-iam";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an IAM service client object.
const iamClient = new IAMClient({ region: REGION });
export { iamClient };
```

This example code can be found here on GitHub.

Create a Node.js module with the file name `iam_deleteservercert.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. Create a JSON object containing the parameters needed to delete a server certificate, which consists of the name of the certificate to delete. Call the `DeleteServerCertificatesCommand` method of the `IAM` client service object.

> **Note**
> Replace *`CERTIFICATE_NAME`* with the name of the server certificate to delete.

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

// Set the parameters
const params = { ServerCertificateName: "CERTIFICATE_NAME" }; // CERTIFICATE_NAME

const run = async () => {
  try {
    const data = await iamClient.send(
      new DeleteServerCertificateCommand(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 iam_deleteservercert.js
```

This example code can be found here on GitHub.

Managing IAM account aliases

This Node.js code example shows:

- How to manage aliases for your AWS account ID.

The scenario

If you want the URL for your sign-in page to contain your company name or other friendly identifier instead of your AWS account ID, you can create an alias for your AWS account ID. If you create an AWS account alias, your sign-in page URL changes to incorporate the alias.

In this example, a series of Node.js modules are used to create and manage IAM account aliases. The Node.js modules use the SDK for JavaScript to manage aliases using these methods of the `IAM` client class:

- `CreateAccountAliasCommand`
- `ListAccountAliasesCommand`
- `DeleteAccountAliasCommand`

For more information about IAM account aliases, see Your AWS account ID and its alias in the *IAM User 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 Loading credentials in Node.js from the shared credentials file (p. 33).

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 (p. 56).

Creating an account alias

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

```
import { IAMClient } from "@aws-sdk/client-iam";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an IAM service client object.
const iamClient = new IAMClient({ region: REGION });
export { iamClient };
```

This example code can be found here on GitHub.

Create a Node.js module with the file name `iam_createaccountalias.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. Create a JSON object containing the parameters needed to create an account alias, which includes the alias to create. Call the `CreateAccountAliasCommand` method of the `IAM` client service object.

Note

Replace *`ACCOUNT_ALIAS`* with the alias to create.

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

// Set the parameters
const params = { AccountAlias: "ACCOUNT_ALIAS" }; //ACCOUNT_ALIAS

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

```
    const data = await iamClient.send(new CreateAccountAliasCommand(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 iam_createaccountalias.js
```

This example code can be found here on GitHub.

Listing account aliases

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

```
import { IAMClient } from "@aws-sdk/client-iam";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an IAM service client object.
const iamClient = new IAMClient({ region: REGION });
export { iamClient };
```

This example code can be found here on GitHub.

Create a Node.js module with the file name `iam_listaccountaliases.js`. Be sure to configure the SDK as client previously shown, including downloading the required clients and packages. Create a JSON object containing the parameters needed to list account aliases, which includes the maximum number of items to return. Call the `ListAccountAliasesCommand` method of the `IAM` client service object.

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

// Set the parameters
const params = { MaxItems: 5 };

const run = async () => {
  try {
    const data = await iamClient.send(new ListAccountAliasesCommand(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 iam_listaccountaliases.js
```

This example code can be found here on GitHub.

Deleting an account alias

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

```
import { IAMClient } from "@aws-sdk/client-iam";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an IAM service client object.
const iamClient = new IAMClient({ region: REGION });
export { iamClient };
```

This example code can be found here on GitHub.

Create a Node.js module with the file name `iam_deleteaccountalias.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. Create a JSON object containing the parameters needed to delete an account alias, which includes the alias you want deleted. Call the `DeleteAccountAliasCommand` method of the `IAM` service object.

Note
Replace *ALIAS* with the name of the alias you want to delete.

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

// Set the parameters
const params = { AccountAlias: "ALIAS" }; // ALIAS

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

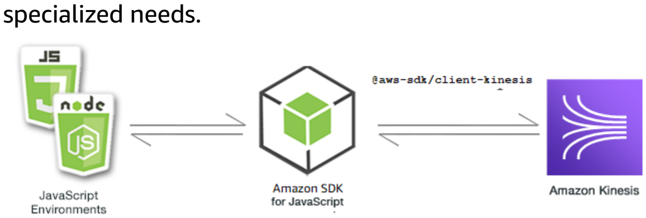
To run the example, enter the following at the command prompt.

```
node iam_deleteaccountalias.js
```

This example code can be found here on GitHub.

Amazon Kinesis Examples

Amazon Kinesis is a platform for streaming data on AWS, offering powerful services to load and analyze streaming data, and also providing the ability for you to build custom streaming data applications for specialized needs.



The JavaScript API for Kinesis is exposed through the `Kinesis` client class. For more information about using the Kinesis client class, see Class: Kinesis in the API Reference.

Topics

Capturing Webpage Scroll Progress with Amazon Kinesis

In this example, a simple HTML page simulates the content of a blog page. As the reader scrolls the simulated blog post, the browser script uses the SDK for JavaScript to record the scroll distance down the page and send that data to Kinesis using the `PutRecordsCommand` method of the Kinesis client class. The streaming data captured by Amazon Kinesis Data Streams can then be processed by Amazon EC2 instances and stored in any of several data stores including Amazon DynamoDB and Amazon Redshift.

To build the app:

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, including *webpack*. Follow the instructions on GitHub.
- Create a shared configurations file with your user credentials. For more information about providing a credentials JSON file, see Loading credentials in Node.js from the shared credentials file (p. 33).

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 (p. 56).

Create the AWS resources

This topic is part of a example that demonstrates how to capture and process browser event data with Amazon Kinesis. To start at the beginning of the example, see Capturing Webpage Scroll Progress with Amazon Kinesis (p. 163).

This example requires the following resources.

- An Amazon Kinesis stream.
- An Amazon Cognito identity pool with access enabled for unauthenticated identities.
- An AWS Identity and Access Management role whose policy grants permission to submit data to an Amazon Kinesis stream.

You can create these resources manually, but we recommend provisioning these resources using the AWS CloudFormation as described in this topic.

Create the AWS resources using the AWS CloudFormation

To create the 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 developer guide..

1. Install and configure the AWS CloudFormation 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 (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.

   Note
   If you create anoher stack using this template, you must change the stream name in the `setup.yaml` or you encounter an error.

To view the resources when they are created, to the the **Stacks** page on the AWS AWS CloudFormation console, select the stack, and open the **Resources** tab.

You require the following for this example:

- An Amazon Kinesis stream. You need to include the name of the stream the browser script.
- An Amazon Cognito identity pool with access enabled for unauthenticated identities. You need to include the identity pool ID in the code to obtain credentials for the browser script. For more information about Amazon Cognito identity pools, see Identity Pools in the *Amazon Cognito Developer Guide*.
- An IAM role with an attached IAM policy that grants permission to submit data to an Amazon Kinesis stream. For more information about creating an IAM role, see Creating a Role to Delegate Permissions to an AWS Service in the *IAM User Guide*.

Note
This is the role policy when is attached to the IAM role. The CDK automatically poplulates the *STREAM_RESOURCE_ARN*.

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

Note
The CDK automatically poplulates the *STREAM_RESOURCE_ARN*.

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.

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.

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 example.

Create the Blog page in HTML

The HTML for the blog page consists mainly of a series of paragraphs contained within a `<div>` element. The scrollable height of this `<div>` is used to help calculate how far a reader has scrolled through the content as they read. The HTML also contains a `<script>` element which adds the `main.js`.This file contains the browser script that captures scroll progress on the page and reports it to Kinesis and the required AWS SDK for JavaScript modules. You create this script using *webpack*, as described in the Bundling the browser script (p. 168) section of this example.

```
<!DOCTYPE html>
<html>
<head>
    <title>AWS SDK for JavaScript (V3) - Amazon Kinesis Application</title>
```

```
</head>
<body>
<div id="BlogContent" style="width: 60%; height: 800px; overflow: auto;margin: auto; text-
align: center;">
    <div>
        <p>
            Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vestibulum vitae nulla
 eget nisl bibendum feugiat. Fusce rhoncus felis at ultricies luctus. Vivamus fermentum
 cursus sem at interdum. Proin vel lobortis nulla. Aenean rutrum odio in tellus semper
 rhoncus. Nam eu felis ac augue dapibus laoreet vel in erat. Vivamus vitae mollis turpis.
 Integer sagittis dictum odio. Duis nec sapien diam. In imperdiet sem nec ante laoreet,
 vehicula facilisis sem placerat. Duis ut metus egestas, ullamcorper neque et, accumsan
 quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos
 himenaeos.
        </p>
        <!-- Additional paragraphs in the blog page appear here -->
    </div>
</div>
<script src="main.js"></script>
</body>
</html>
```

Create the browser script

Configuring the SDK

Create a `libs` directory, and create a Node.js module with the file name `kinesisClient.js`. Copy and paste the code below into it, which creates the Kinesis client object. Replace ***REGION*** with your AWS Region. Replace ***IDENTITY_POOL_ID*** with the Amazon Cognito identity pool id you created in Create the AWS resources (p. 163).

```
// Import the required AWS SDK clients and commands for Node.js.
const {CognitoIdentityClient} = require("@aws-sdk/client-cognito-identity");
const {
  fromCognitoIdentityPool,
} = require("@aws-sdk/credential-provider-cognito-identity");
const { KinesisClient, PutRecordsCommand } = require("@aws-sdk/client-kinesis");

// Configure Credentials to use Cognito
const REGION = "REGION";
const kinesisClient = new KinesisClient({
  region: REGION,
  credentials: fromCognitoIdentityPool({
    client: new CognitoIdentityClient({region: REGION}),
    identityPoolId: "IDENTITY_POOL_ID" // IDENTITY_POOL_ID
  })
});
export {kinesisClient}
```

You can find this code here on GitHub.

Creating Scroll Records

Scroll progress is calculated using the `scrollHeight` and `scrollTop` properties of the `<div>` containing the content of the blog post. Each scroll record is created in an event listener function for the `scroll` event and then added to an array of records for periodic submission to Kinesis. Replace ***PARTITION_KEY*** with a paritition key, which must be a string. For more information about partition strings, see PutRecord in the *Amazon Kinesis Data Analytics developer guide*.

The following code snippet shows this step. (See Bundling the browser script (p. 168) for the full example.)

```
import { PutRecordsCommand } from "@aws-sdk/client-kinesis";
import { kinesisClient } from "./libs/kinesisClient.js";
// Get the ID of the web page element.
var blogContent = document.getElementById('BlogContent');

// Get scrollable height.
var scrollableHeight = blogContent.clientHeight;

var recordData = [];
var TID = null;
blogContent.addEventListener('scroll', function(event) {
  console.log('scrolled');
  clearTimeout(TID);
  // Prevent creating a record while a user is actively scrolling.
  TID = setTimeout(function() {
    // Calculate the percentage.
    var scrollableElement = event.target;
    var scrollHeight = scrollableElement.scrollHeight;
    var scrollTop = scrollableElement.scrollTop;

    var scrollTopPercentage = Math.round((scrollTop / scrollHeight) * 100);
    var scrollBottomPercentage = Math.round(((scrollTop + scrollableHeight) / scrollHeight)
 * 100);

    // Create the Amazon Kinesis record.
    var record = {
      Data: JSON.stringify({
        blog: window.location.href,
        scrollTopPercentage: scrollTopPercentage,
        scrollBottomPercentage: scrollBottomPercentage,
        time: new Date()
      }),
      PartitionKey: 'PARTITION_KEY' // Must be a string.
    };
    recordData.push(record);
  }, 100);
});
```

Submitting Records to Kinesis

Once each second, if there are records in the array, those pending records are sent to Kinesis. Replace *STREAM_NAME* with the name of the stream you created in the Create the AWS resources (p. 163) section of this example.

> **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 (p. 3).

The following code snippet shows this step. (See Bundling the browser script (p. 168) for the full example.)

```
// Helper function to upload data to Amazon Kinesis.
const uploadData = async () => {
  try {
    const data = await client.send(new PutRecordsCommand({
      Records: recordData,
      StreamName: 'STREAM_NAME'
    }));
    console.log('data', data);
    console.log("Kinesis updated", data);
  } catch (err) {
    console.log("Error", err);
```

```
  }
};
// Run uploadData every second if data exists.
setInterval(function() {
  if (!recordData.length) {
    return;
  }
  uploadData();
  // clear record data
  recordData = [];
}, 1000);
```

Bundling the browser script

This topic describes how to bundle the browser script that captures scroll progress on the page and reports it to Kinesis and the required AWS SDK for JavaScript modules for this example.

If you haven't already, follow the Prerequisite tasks (p. 163) for this example to install webpack.

> **Note**
> For information about*webpack*, see Bundling applications with webpack (p. 42).

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

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

Here is the complete browser script code for the Kinesis capturing webpage scroll progress example.

```
// Configure Credentials to use Cognito
const { CognitoIdentityClient } = require("@aws-sdk/client-cognito-identity");
const {
  fromCognitoIdentityPool,
} = require("@aws-sdk/credential-provider-cognito-identity");
const { Kinesis, PutRecordsCommand } = require("@aws-sdk/client-kinesis");

const REGION = "REGION";
const client = new Kinesis({
  region: REGION,
  credentials: fromCognitoIdentityPool({
    client: new CognitoIdentityClient({region: REGION}),
    identityPoolId: "IDENTITY_POOL_ID" // IDENTITY_POOL_ID
  })
});
// Get the ID of the web page element.
var blogContent = document.getElementById('BlogContent');

// Get scrollable height.
var scrollableHeight = blogContent.clientHeight;

var recordData = [];
var TID = null;
blogContent.addEventListener('scroll', function(event) {
  console.log('scrolled');
  clearTimeout(TID);
  // Prevent creating a record while a user is actively scrolling.
  TID = setTimeout(function() {
    // Calculate the percentage.
    var scrollableElement = event.target;
    var scrollHeight = scrollableElement.scrollHeight;
    var scrollTop = scrollableElement.scrollTop;
```

```
    var scrollTopPercentage = Math.round((scrollTop / scrollHeight) * 100);
    var scrollBottomPercentage = Math.round(((scrollTop + scrollableHeight) / scrollHeight)
 * 100);

    // Create the Amazon Kinesis record.
    var record = {
      Data: JSON.stringify({
        blog: window.location.href,
        scrollTopPercentage: scrollTopPercentage,
        scrollBottomPercentage: scrollBottomPercentage,
        time: new Date()
      }),
      PartitionKey: 'PARTITION_KEY' // Must be a string.
    };
    recordData.push(record);
  }, 100);
});

// Helper function to upload data to Amazon Kinesis.
const uploadData = async () => {
  try {
    const data = await client.send(new PutRecordsCommand({
      Records: recordData,
      StreamName: 'STREAM_NAME'
    }));
    console.log('data', data);
    console.log("Kinesis updated", data);
  } catch (err) {
    console.log("Error", err);
  }
};
// Run uploadData every second if data exists.
setInterval(function() {
  if (!recordData.length) {
    return;
  }
  uploadData();
  // clear record data
  recordData = [];
}, 1000);
```

Run the example

This topic is part of an example that demonstrates how to capture and process browser event data with Amazon Kinesis. To start at the beginning of the example, see Capturing Webpage Scroll Progress with Amazon Kinesis (p. 163).

Open the `blog_page.html` in your browser. Adjust the size of the browser window until scroll-bars display around the text. When you scroll the scroll-bars, the event data is caputred in Kinesis. To view the recorded data, open the stream in the Amazon Command Console, choose the **Monitoring** tab, and examine the information on the **Stream metrics** pane.

Delete the resources

This topic is part of a example that demonstrates how to capture and process browser event data with Amazon Kinesis. To start at the beginning of the example, see Capturing Webpage Scroll Progress with Amazon Kinesis (p. 163).

When you finish the example, you should delete the resources so you do not incur any unnecessary charges. You can do this by following either Deleting a stack in the Amazon Web Services Management Console or the Deleting a stack in the command line.

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.

> **Important**
> AWS SDK for JavaScript v2, and not v3, is installed in the default Lambda Node.js environment. The examples linked here demonstrate how to bundle the required AWS SDK for JavaScript v3 modules with the example code.

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:

- Creating and using Lambda functions (p. 342) (Simplified example)
- Invoking Lambda with API Gateway (p. 312)
- Creating scheduled events to execute AWS Lambda functions (p. 334)

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 (p. 349)

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 Loading credentials in Node.js from the shared credentials file (p. 33).
- 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 (p. 171)section below describes how to create these resources.

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

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

Create the AWS resources using the AWS CloudFormation

AWS CloudFormation enables you to create and provision AWS infrastructure deployments predictably and repeatedly. For more information about AWS CloudFormation, see the AWS CloudFormation developer 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.

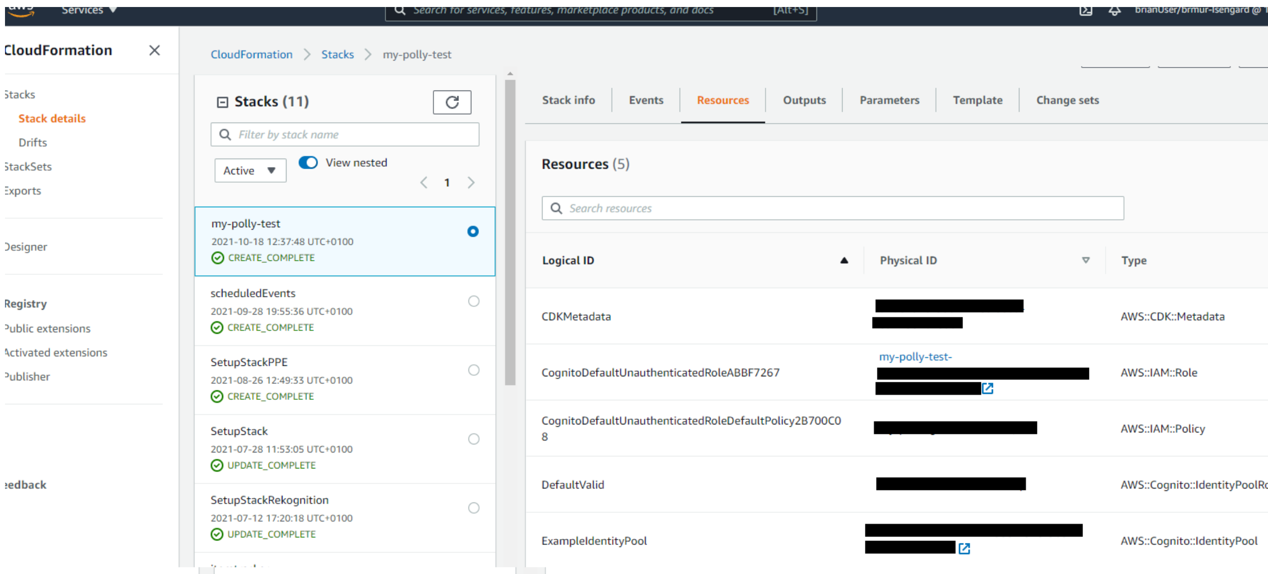
   Note
   The AWS CloudFormation template was generated using the AWS CDK available here on GitHub. For more information about the AWS CDK, see the AWS Cloud Development Kit (CDK) Developer Guide.
3. Run the following command from the command line, replacing *`STACK_NAME`* with a unique name for the stack.

   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 synthesize the voice message and upload it to the Amazon S3 bucket.

```
const {
  Polly,
  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",
};
```

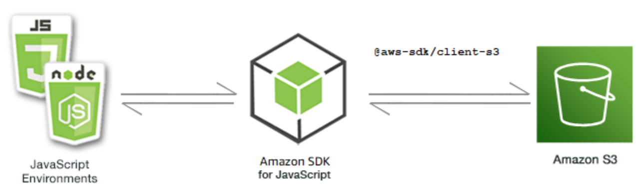
```
const run = async () => {
  try {
    const data = await pollyClient.send(
      new StartSpeechSynthesisTaskCommand(params)
    );
    console.log("Success, audio file added to " + params.OutputS3BucketName);
  } catch (err) {
    console.log("Error putting object", err);
  }
};
run();
```

This sample code can be found here on GitHub.

Amazon S3 examples

Amazon Simple Storage Service (Amazon S3) is a web service that provides highly scalable cloud storage. Amazon S3 provides easy to use object storage, with a simple web service interface to store and retrieve any amount of data from anywhere on the web.

Topics



The JavaScript API for Amazon S3 is exposed through the `S3` client class. For more information about using the Amazon S3 client class, see Class: S3 in the API Reference.

Amazon S3 browser examples

The following topics show two examples of how the AWS SDK for JavaScript can be used in the browser to interact with Amazon S3 buckets.

- The first shows a simple scenario in which the existing photos in an Amazon S3 bucket can be viewed by any (unauthenticated) user.
- The second shows a more complex scenario in which users are allowed to perform operations on photos in the bucket such as upload, delete, and so on.

Topics

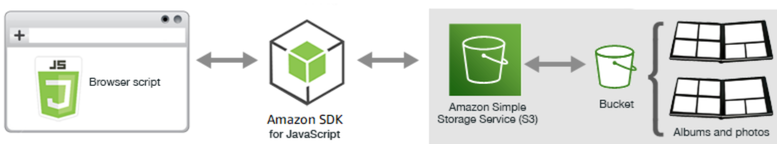
Viewing photos in an Amazon S3 bucket from a browser

This browser script code example shows:

- How to create a photo album in an Amazon Simple Storage Service (Amazon S3) bucket and allow unauthenticated users to view the photos.

The scenario

In this example, a simple HTML page provides a browser-based application for viewing the photos in a photo album. The photo album is in an Amazon S3 bucket into which photos are uploaded.



The browser script uses the SDK for JavaScript to interact with an Amazon S3 bucket. The script uses the `ListObjectsCommand` method of the Amazon S3 client class to enable you to view the photo albums.

Prerequisite tasks

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

> **Note**
> In this example, you must use the same AWS Region for both the Amazon S3 bucket and the Amazon Cognito identity pool.

Set up your local environment

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 the bucket

In the Amazon S3 console, create an Amazon S3 bucket where you can store albums and photos. For more information about using the console to create an S3 bucket, see Creating a bucket in the *Amazon Simple Storage Service User Guide*.

As you create the Amazon S3 bucket, be sure to do the following:

- Make note of the bucket name so you can use it in a subsequent prerequisite task, Configure role permissions (p. 175).
- Choose an AWS Region to create the bucket in. This must be the same Region that you'll use to create an Amazon Cognito identity pool in a subsequent prerequisite task, Create an identity pool (p. 175).
- In the **Create Bucket** wizard, on the **Create Bucket** page, in the **Bucket settings for block public access** section, clear these boxes: **Block public access to buckets and objects granted through new access control lists (ACLs)** and **Block public access to buckets and objects granted through any access control lists (ACLs)**.

For information about how to check and configure bucket permissions, see Setting permissions for website access in the *Amazon Simple Storage Service User Guide*.

Create an identity pool

On the Amazon Cognito console, create an Amazon Cognito identity pool.

As you create the identity pool:

- Make note of the identity pool name, and the role name for the **unauthenticated** identity.
- On the **Sample Code** page, select "JavaScript" from the **Platform** list. Then copy or write down the sample code.

  Note
  You must choose "JavaScript" from the **Platform** list for your code to work.

Configure role permissions

To allow viewing of albums and photos, you have to add permissions to an IAM role of the identity pool that you just created. Start by creating a policy as follows.

1. Open the IAM console.
2. In the navigation pane on the left, choose **Policies**, and then choose **Create policy**.
3. On the **JSON** tab, enter the following JSON definition, but replace *BUCKET_NAME* with the name of the bucket.

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "s3:ListBucket"
         ],
         "Resource": [
            "arn:aws:s3:::BUCKET_NAME"
         ]
      }
   ]
}
```

4. Choose the **Review policy** button, name the policy and provide a description (if you want), and then choose the **Create policy** button.

   Be sure to make note of the name so that you can find it and attach it to the IAM role later.

After the policy is created, navigate back to the IAM console. Find the IAM role for the **unauthenticated** identity that Amazon Cognito created in the previous prerequisite task. You use the policy you just created to add permissions to this identity.

For additional information about creating an IAM role, see Creating a role to delegate permissions to an AWS service in the *IAM User Guide*.

Configure CORS

Before the browser script can access the Amazon S3 bucket, you have to set up its CORS configuration (p. 41) as follows.

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

JSON

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

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<CORSConfiguration xmlns="http://s3.amazonaws.com/doc/2006-03-01/">
    <CORSRule>
        <AllowedOrigin>*</AllowedOrigin>
        <AllowedMethod>GET</AllowedMethod>
        <AllowedMethod>HEAD</AllowedMethod>
        <AllowedHeader>*</AllowedHeader>
    </CORSRule>
</CORSConfiguration>
```

Create albums and upload photos

Because this example only allows users to view the photos that are already in the bucket, you need to create some albums in the bucket and upload photos to them.

Note
For this example, the file names of the photo files must start with a single underscore ("_"). This character is important later for filtering. In addition, be sure to respect the copyrights of the owners of the photos.

1. On the Amazon S3 console, open the bucket that you created earlier.
2. On the **Overview** tab, choose **Create folder** to create folders. For this example, name the folders "album1", "album2", and "album3".
3. For **album1** and then **album2**, select the folder and then upload photos to it as follows:
   a. Choose **Upload**.
   b. Drag or choose the photo files you want to use, and then choose **Next**.
   c. Under **Manage public permissions**, choose **Grant public read access to this object(s)**.
   d. Choose **Upload** (in the lower-left corner).
4. Leave **album3** empty.

Install the required SDK clients and packages

Install the following SDK modules:

- `client-s3`
- `client-cognito-identity`
- `credential-provider-cognito-identity`

> **Note**
> For information on installing SDK modules, see Installing the SDK for JavaScript (p. 24).

Install webpack

To use V3 of the AWS SDK for JavaScript in the browser, you require `webpack` to bundle the Javascript modules and functions.

To install `webpack`, run the following at a command prompt.

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

> **Important**
> To view a sample of the `package.json`for this example, see the AWS SDK for JavaScript code samples on GitHub.
>
> **Note**
> For information on installing Webpack, see Bundling applications with webpack (p. 42).

Defining the webpage

The HTML for the photo-viewing application consists of a `<div>` element in which the browser script creates the viewing interface.

The `<script>` element adds the `main.js` file, which contains all the required JavaScript for the example.

> **Note**
> To generate the `main.js` file, see Running the code (p. 180).
>
> **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 (p. 3).

```
<!DOCTYPE html>
<html>
<head>
    <script type="text/javascript" src="./main.js"></script>
</head>
<body>
<h1>Photo album viewer</h1>
<div id="viewer" />
<script>
    listAlbums();
</script>
</body>
</html>
```

Configuring the SDK

Obtain the credentials you need to configure the SDK by calling the `CognitoIdentityCredentials` method. You need to provide the Amazon Cognito identity pool ID. Then create an `S3` service object.

```
// Load the required clients and packages
const { CognitoIdentityClient } = require("@aws-sdk/client-cognito-identity");
const {
  fromCognitoIdentityPool,
} = require("@aws-sdk/credential-provider-cognito-identity");
const { S3Client, ListObjectsCommand } = require("@aws-sdk/client-s3");

// Initialize the Amazon Cognito credentials provider
const REGION = "region"; //e.g., 'us-east-1'
const s3 = new S3Client({
  region: REGION,
  credentials: fromCognitoIdentityPool({
    client: new CognitoIdentityClient({ region: REGION }),
    identityPoolId: "IDENTITY_POOL_ID", // IDENTITY_POOL_ID e.g., eu-west-1:xxxxxx-xxxx-
xxxx-xxxx-xxxxxxxxx
  }),
});
```

The remaining code in this example defines the following functions to gather and present information about the albums and photos in the bucket.

- `listAlbums`
- `viewAlbum`

Listing albums in the bucket

To list all of the existing albums in the bucket, the application's `listAlbums` function calls the `ListObjectsCommand` method of the `S3` client service object. The function uses the `CommonPrefixes` property so that the call returns only objects that are used as albums (that is, the folders).

The rest of the function takes the list of albums from the Amazon S3 bucket and generates the HTML needed to display the album list on the webpage.

> **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 (p. 3).

```
// A utility function to create HTML.
function getHtml(template) {
  return template.join("\n");
}
// Make the getHTML function available to the browser
window.getHTML = getHtml;

// List the photo albums that exist in the bucket
var albumBucketName = "BUCKET_NAME"; //BUCKET_NAME

const listAlbums = async () => {
  try {
    const data = await s3.send(
      new ListObjectsCommand({ Delimiter: "/", Bucket: albumBucketName })
    );
    var albums = data.CommonPrefixes.map(function (commonPrefix) {
      var prefix = commonPrefix.Prefix;
      var albumName = decodeURIComponent(prefix.replace("/", ""));
      return getHtml([
        "<li>",
        '<button style="margin:5px;" onclick="viewAlbum(\'' +
```

```
          albumName +
          "')\">",
        albumName,
        "</button>",
        "</li>",
      ]);
    });
    var message = albums.length
      ? getHtml(["<p>Click an album name to view it.</p>"])
      : "<p>You don't have any albums. You need to create an album.";
    var htmlTemplate = [
      "<h2>Albums</h2>",
      message,
      "<ul>",
      getHtml(albums),
      "</ul>",
    ];
    document.getElementById("viewer").innerHTML = getHtml(htmlTemplate);
  } catch (err) {
    return alert("There was an error listing your albums: " + err.message);
  }
};
// Make the viewAlbum function available to the browser
window.listAlbums = listAlbums;
```

Viewing an album

To display the contents of an album in the Amazon S3 bucket, the application's `viewAlbum` function takes an album name and creates the Amazon S3 key for that album. The function then calls the `ListObjectsCommand` method of the `S3` client service object to obtain a list of all the objects (the photos) in the album.

The rest of the function takes the list of objects that are in the album and generates the HTML needed to display the photos on the webpage.

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 (p. 3).

```
// Show the photos that exist in an album
const viewAlbum = async (albumName) => {
  try {
    var albumPhotosKey = encodeURIComponent(albumName) + "/";
    const data = await s3.send(
      new ListObjectsCommand({
        Prefix: albumPhotosKey,
        Bucket: albumBucketName,
      })
    );
    var href = "https://s3." + REGION + ".amazonaws.com/";
    var bucketUrl = href + albumBucketName + "/";
    var photos = data.Contents.map(function (photo) {
      var photoKey = photo.Key;
      var photoUrl = bucketUrl + encodeURIComponent(photoKey);
      return getHtml([
        "<span>",
        "<div>",
        "<br/>",
        '<img style="width:128px;height:128px;" src="' + photoUrl + '"/>',
        "</div>",
```

```
          "<div>",
          "<span>",
          photoKey.replace(albumPhotosKey, ""),
          "</span>",
          "</div>",
          "</span>",
        ]);
      });
      var message = photos.length
        ? "<p>The following photos are present.</p>"
        : "<p>There are no photos in this album.</p>";
      var htmlTemplate = [
        "<div>",
        '<button onclick="listAlbums()">',
        "Back To albums",
        "</button>",
        "</div>",
        "<h2>",
        "Album: " + albumName,
        "</h2>",
        message,
        "<div>",
        getHtml(photos),
        "</div>",
        "<h2>",
        "End of album: " + albumName,
        "</h2>",
        "<div>",
        '<button onclick="listAlbums()">',
        "Back To albums",
        "</button>",
        "</div>",
      ];
      document.getElementById("viewer").innerHTML = getHtml(htmlTemplate);
      document
        .getElementsByTagName("img")[0]
        .setAttribute("style", "display:none;");
    } catch (err) {
      return alert("There was an error viewing your album: " + err.message);
    }
  };

// Make the viewAlbum function available to the browser
window.viewAlbum = viewAlbum;
```

Running the code

To run the code for this example

1. Save all the code as `s3_PhotoViewer.js`.

   Note
   This file is available on GitHub.

2. Replace *"REGION"* with your AWS Region, such as `us-west-2`.
3. Replace *"BUCKET_NAME"* with your Amazon S3 bucket.
4. Replace *"IDENTITY_POOL_ID"* with the `IdentityPoolId` from the **Sample page** of the Amazon Cognito identity pool you created for this example.
5. Run the following at the command prompt to bundle the JavaScript for this example into a file called `main.js`.

```
webpack s3_PhotoViewer.js --mode development --target web --devtool false -o main.js
```

Note
For information about installing `webpack`, see Bundling applications with webpack (p. 42).

6. Run the following in the command line:

```
node s3_PhotoViewer.js
```

Viewing photos in an Amazon S3 bucket: Full code

This section contains the full HTML and JavaScript code for the example in which photos in an Amazon S3 bucket can be viewed. See the the Prerequisites section (p. 174) for details and prerequisites.

The HTML for the example:

```
<!DOCTYPE html>
<html>
<head>
    <script type="text/javascript" src="./main.js"></script>
</head>
<body>
<h1>Photo album viewer</h1>
<div id="viewer" />
<script>
    listAlbums();
</script>
</body>
</html>
```

This example code can be found here on GitHub.

The following is the browser script code for the example.

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 (p. 3).

```
// Load the required clients and packages
const { CognitoIdentityClient } = require("@aws-sdk/client-cognito-identity");
const {
  fromCognitoIdentityPool,
} = require("@aws-sdk/credential-provider-cognito-identity");
const { S3Client, ListObjectsCommand } = require("@aws-sdk/client-s3");

// Initialize the Amazon Cognito credentials provider
const REGION = "region"; //e.g., 'us-east-1'
const s3 = new S3Client({
  region: REGION,
  credentials: fromCognitoIdentityPool({
    client: new CognitoIdentityClient({ region: REGION }),
    identityPoolId: "IDENTITY_POOL_ID", // IDENTITY_POOL_ID e.g., eu-west-1:xxxxxx-xxxx-
xxxx-xxxx-xxxxxxxxxx
  }),
});

// A utility function to create HTML.
function getHtml(template) {
  return template.join("\n");
}
// Make the getHTML function available to the browser
window.getHTML = getHtml;
```

```
// List the photo albums that exist in the bucket
var albumBucketName = "BUCKET_NAME"; //BUCKET_NAME

const listAlbums = async () => {
  try {
    const data = await s3.send(
      new ListObjectsCommand({ Delimiter: "/", Bucket: albumBucketName })
    );
    var albums = data.CommonPrefixes.map(function (commonPrefix) {
      var prefix = commonPrefix.Prefix;
      var albumName = decodeURIComponent(prefix.replace("/", ""));
      return getHtml([
        "<li>",
        '<button style="margin:5px;" onclick="viewAlbum(\'' +
          albumName +
          "')\">",
        albumName,
        "</button>",
        "</li>",
      ]);
    });
    var message = albums.length
      ? getHtml(["<p>Click an album name to view it.</p>"])
      : "<p>You don't have any albums. You need to create an album.";
    var htmlTemplate = [
      "<h2>Albums</h2>",
      message,
      "<ul>",
      getHtml(albums),
      "</ul>",
    ];
    document.getElementById("viewer").innerHTML = getHtml(htmlTemplate);
  } catch (err) {
    return alert("There was an error listing your albums: " + err.message);
  }
};
// Make the viewAlbum function available to the browser
window.listAlbums = listAlbums;


// Show the photos that exist in an album
const viewAlbum = async (albumName) => {
  try {
    var albumPhotosKey = encodeURIComponent(albumName) + "/";
    const data = await s3.send(
      new ListObjectsCommand({
        Prefix: albumPhotosKey,
        Bucket: albumBucketName,
      })
    );
    var href = "https://s3." + REGION + ".amazonaws.com/";
    var bucketUrl = href + albumBucketName + "/";
    var photos = data.Contents.map(function (photo) {
      var photoKey = photo.Key;
      var photoUrl = bucketUrl + encodeURIComponent(photoKey);
      return getHtml([
        "<span>",
        "<div>",
        "<br/>",
        '<img style="width:128px;height:128px;" src="' + photoUrl + '"/>',
        "</div>",
        "<div>",
        "<span>",
        photoKey.replace(albumPhotosKey, ""),
        "</span>",
```

```
        "</div>",
        "</span>",
      ]);
    });
    var message = photos.length
      ? "<p>The following photos are present.</p>"
      : "<p>There are no photos in this album.</p>";
    var htmlTemplate = [
      "<div>",
      '<button onclick="listAlbums()">',
      "Back To albums",
      "</button>",
      "</div>",
      "<h2>",
      "Album: " + albumName,
      "</h2>",
      message,
      "<div>",
      getHtml(photos),
      "</div>",
      "<h2>",
      "End of album: " + albumName,
      "</h2>",
      "<div>",
      '<button onclick="listAlbums()">',
      "Back To albums",
      "</button>",
      "</div>",
    ];
    document.getElementById("viewer").innerHTML = getHtml(htmlTemplate);
    document
      .getElementsByTagName("img")[0]
      .setAttribute("style", "display:none;");
  } catch (err) {
    return alert("There was an error viewing your album: " + err.message);
  }
};

// Make the viewAlbum function available to the browser
window.viewAlbum = viewAlbum;
```

This example code can be found here on GitHub.

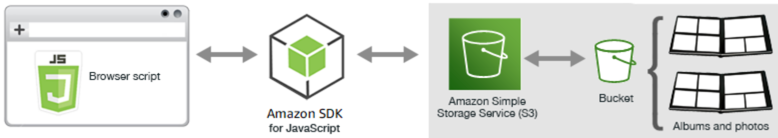
Uploading photos to Amazon S3 from a browser

This browser script code example shows:

- How to create a browser application that allows users to create photo albums in an Amazon S3 bucket and upload photos into the albums.

The scenario

In this example, a simple HTML page provides a browser-based application for creating photo albums in an Amazon S3 bucket into which you can upload photos. The application lets you delete photos and albums that you add.



The browser script uses the SDK for JavaScript to interact with an Amazon S3 bucket. Use the following methods of the Amazon S3 client class to enable the photo album application:

- `PutObjectCommand`
- `DeleteObjectCommand`
- `ListObjectsCommand`

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.
- In the Amazon S3 console, create an Amazon S3 bucket that you will use to store the photos in the album. For more information about creating a bucket in the console, see Creating a bucket in the *Amazon Simple Storage Service User Guide*. Make sure you have both **Read** and **Write** permissions on **Objects**. For more information about setting bucket permissions, see Setting permissions for website access.
- In the Amazon Cognito console, create an Amazon Cognito identity pool using Federated Identities with access enabled for unauthenticated users in the same Region as the Amazon S3 bucket. You need to include the identity pool ID in the code to obtain credentials for the browser script. For more information about Amazon Cognito Federated Identities, see Amazon Cognito identity pools (federated identites) in the *Amazon Cognito Developer Guide*.
- In the IAM console, find the IAM role created by Amazon Cognito for unauthenticated users. Add the following policy to grant read and write permissions to an Amazon S3 bucket. For more information about creating an IAM role, see Creating a role to delegate permissions to an AWS service in the *IAM User Guide*.

  Use this role policy for the IAM role created by Amazon Cognito for unauthenticated users.

  > **Warning**
  > If you enable access for unauthenticated users, you will grant write access to the bucket, and all objects in the bucket, to anyone in the world. This security posture is useful in this example to keep it focused on the primary goals of the example. In many live situations, however, tighter security, such as using authenticated users and object ownership, is highly advisable.

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "s3:DeleteObject",
            "s3:GetObject",
            "s3:ListBucket",
            "s3:PutObject",
            "s3:PutObjectAcl"
         ],
         "Resource": [
            "arn:aws:s3:::BUCKET_NAME",
            "arn:aws:s3:::BUCKET_NAME/*"
```

```
            ]
        }
    ]
}
```

Configuring CORS

Before the browser script can access the Amazon S3 bucket, you must first set up its CORS configuration (p. 41) as follows.

> **Important**
> In Amazon S3 on the new AWS Web Services Management console, the CORS configuration must be JSON.

JSON

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

XML

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

Install the required SDK clients and packages

Install the following SDK modules:

> **Note**
> For information on installing SDK modules, see Installing the SDK for JavaScript (p. 24).

- client-s3
- client-cognito-identity
- credential-provider-cognito-identity

Install webpack

To use V3 of the AWS SDK for JavaScript in the browser, you require Webpack to bundle the javascript modules and functions.

To install web pack, run the following in the command line:

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

> **Important**
> To view a sample of the package.json for this example, see the AWS SDK for JavaScript code samples on GitHub.

> **Note**
> For information on installing Webpack, see Bundling applications with webpack (p. 42).

Defining the webpage

The HTML for the photo-viewing application consists of a `<div>` element in which the browser script creates the viewing/uploading interface.

The `<script>` element adds the `<main.js>` file, which contains all the required JavaScript for the example.

> **Note**
> To generate the `<main.js>` file, see Running the code (p. 192) below.

> **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 (p. 3).

```
<!DOCTYPE html>
<html>
<head>
    <script src="./main.js"></script>
    <script>
        function getHtml(template) {
            return template.join("\n");
        }
        listAlbums();
    </script>
</head>
<body>
<h1>My photo albums app</h1>
<div id="app"></div>
</body>
</html>
```

Configuring the SDK

Obtain the credentials needed to configure the SDK by calling the `CognitoIdentityCredentials` method, providing the Amazon Cognito identity pool ID. Next, create an `S3` client service 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 (p. 3).

```
// Load the required clients and packages
const { CognitoIdentityClient } = require("@aws-sdk/client-cognito-identity");
const {
  fromCognitoIdentityPool,
} = require("@aws-sdk/credential-provider-cognito-identity");
const { S3Client, PutObjectCommand, ListObjectsCommand, DeleteObjectCommand,
 DeleteObjectsCommand } = require("@aws-sdk/client-s3");

// Set the AWS Region
const REGION = "REGION"; //REGION

// Initialize the Amazon Cognito credentials provider
const s3 = new S3Client({
  region: REGION,
  credentials: fromCognitoIdentityPool({
    client: new CognitoIdentityClient({ region: REGION }),
    identityPoolId: "IDENTITY_POOL_ID", // IDENTITY_POOL_ID
  }),
});

const albumBucketName = "BUCKET_NAME"; //BUCKET_NAME
```

Nearly all of the rest of the code in this example is organized into a series of functions that gather and present information about the albums in the bucket, upload and display photos uploaded into albums, and delete photos and albums. Those functions are:

- `listAlbums`
- `createAlbum`
- `viewAlbum`
- `addPhoto`
- `deleteAlbum`
- `deletePhoto`

Listing albums in the bucket

The application creates albums in the Amazon S3 bucket as objects whose keys begin with a forward slash character, indicating the object functions as a folder. To list all the existing albums in the bucket, the application's `listAlbums` function calls the `ListObjectsCommand` method of the `S3` client service object while using `commonPrefix` so the call returns only objects used as albums.

The rest of the function takes the list of albums from the Amazon S3 bucket and generates the HTML needed to display the album list in the web page. It also enables deleting and opening individual albums.

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 (p. 3).

```

// A utility function to create HTML
function getHtml(template) {
  return template.join("\n");
}
// Make getHTML function available to the browser
```

```
window.getHTML = getHtml;

// List the photo albums that exist in the bucket
const listAlbums = async () => {
  try {
    const data = await s3.send(
        new ListObjectsCommand({ Delimiter: "/", Bucket: albumBucketName })
    );

    if (data.CommonPrefixes === undefined) {
      const htmlTemplate = [
        "<p>You don't have any albums. You need to create an album.</p>",
        "<button onclick=\"createAlbum(prompt('Enter album name:'))\">",
        "Create new album",
        "</button>",
      ];
      document.getElementById("app").innerHTML = htmlTemplate;
    } else {
      var albums = data.CommonPrefixes.map(function (commonPrefix) {
        var prefix = commonPrefix.Prefix;
        var albumName = decodeURIComponent(prefix.replace("/", ""));
        return getHtml([
          "<li>",
          "<span onclick=\"deleteAlbum('" + albumName + "')\">X</span>",
          "<span onclick=\"viewAlbum('" + albumName + "')\">",
          albumName,
          "</span>",
          "</li>",
        ]);
      });
      var message = albums.length
          ? getHtml([
            "<p>Click an album name to view it.</p>",
            "<p>Click the X to delete the album.</p>",
          ])
          : "<p>You do not have any albums. You need to create an album.";
      const htmlTemplate = [
        "<h2>Albums</h2>",
        message,
        "<ul>",
        getHtml(albums),
        "</ul>",
        "<button onclick=\"createAlbum(prompt('Enter Album Name:'))\">",
        "Create new Album",
        "</button>",
      ];
      document.getElementById("app").innerHTML = getHtml(htmlTemplate);
    }
  } catch (err) {
    return alert("There was an error listing your albums: " + err.message);
  }
};

// Make listAlbums function available to the browser
window.listAlbums = listAlbums;
```

Creating an album in the bucket

To create an album in the Amazon S3 bucket, the application's `createAlbum` function first validates the name given for the new album to ensure it contains suitable characters. The function then forms an Amazon S3 object key, passing it to the `headObject` method of the Amazon S3 service object. This method returns the metadata for the specified key, so if it returns data, then an object with that key already exists.

If the album doesn't already exist, the function calls the `PutObjectCommand` method of the `S3` client service object to create the album. It then calls the `viewAlbum` function to display the new empty album.

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 (p. 3).

```
// Create an album in the bucket
const createAlbum = async (albumName) => {
  albumName = albumName.trim();
  if (!albumName) {
    return alert("Album names must contain at least one non-space character.");
  }
  if (albumName.indexOf("/") !== -1) {
    return alert("Album names cannot contain slashes.");
  }
  var albumKey = encodeURIComponent(albumName);
  try {
    const key = albumKey + "/";
    const params = { Bucket: albumBucketName, Key: key };
    const data = await s3.send(new PutObjectCommand(params));
    alert("Successfully created album.");
    viewAlbum(albumName);
  } catch (err) {
    return alert("There was an error creating your album: " + err.message);
  }
};

// Make createAlbum function available to the browser
window.createAlbum = createAlbum;
```

Viewing an album

To display the contents of an album in the Amazon S3 bucket, the application's `viewAlbum` function takes an album name and creates the Amazon S3 key for that album. The function then calls the `listObjects` method of the `S3` client service object to obtain a list of all the objects (photos) in the album.

The rest of the function takes the list of objects (photos) from the album and generates the HTML needed to display the photos in the web page. It also enables deleting individual photos and navigating back to the album list.

```
// View the contents of an album

const viewAlbum = async (albumName) => {
  const albumPhotosKey = encodeURIComponent(albumName) + "/";
  try {
    const data = await s3.send(
        new ListObjectsCommand({
          Prefix: albumPhotosKey,
          Bucket: albumBucketName,
        })
    );
    if (data.Contents.length === 1) {
      var htmlTemplate = [
        "<p>You don't have any photos in this album. You need to add photos.</p>",
        '<input id="photoupload" type="file" accept="image/*">',
        '<button id="addphoto" onclick="addPhoto(\'' + albumName + "')\">",
```

```
        "Add photo",
        "</button>",
        '<button onclick="listAlbums()">',
        "Back to albums",
        "</button>",
      ];
      document.getElementById("app").innerHTML = getHtml(htmlTemplate);
    } else {
      console.log(data);
      const href = "https://s3." + REGION + ".amazonaws.com/";
      const bucketUrl = href + albumBucketName + "/";
      const photos = data.Contents.map(function (photo) {
        const photoKey = photo.Key;
        console.log(photo.Key);
        const photoUrl = bucketUrl + encodeURIComponent(photoKey);
        return getHtml([
          "<span>",
          "<div>",
          '<img style="width:128px;height:128px;" src="' + photoUrl + '"/>',
          "</div>",
          "<div>",
          "<span onclick=\"deletePhoto('" +
          albumName +
          "','" +
          photoKey +
          "')\">",
          "X",
          "</span>",
          "<span>",
          photoKey.replace(albumPhotosKey, ""),
          "</span>",
          "</div>",
          "</span>",
        ]);
      });
      var message = photos.length
          ? "<p>Click the X to delete the photo.</p>"
          : "<p>You don't have any photos in this album. You need to add photos.</p>";
      const htmlTemplate = [
        "<h2>",
        "Album: " + albumName,
        "</h2>",
        message,
        "<div>",
        getHtml(photos),
        "</div>",
        '<input id="photoupload" type="file" accept="image/*">',
        '<button id="addphoto" onclick="addPhoto(\'' + albumName + "')\">",
        "Add photo",
        "</button>",
        '<button onclick="listAlbums()">',
        "Back to albums",
        "</button>",
      ];
      document.getElementById("app").innerHTML = getHtml(htmlTemplate);
      document.getElementsByTagName("img")[0].remove();
    }
  } catch (err) {
    return alert("There was an error viewing your album: " + err.message);
  }
};
// Make viewAlbum function available to the browser
window.viewAlbum = viewAlbum;
```

Adding photos to an album

To upload a photo to an album in the Amazon S3 bucket, the application's `addPhoto` function uses a file picker element in the web page to identify a file to upload. It then forms a key for the photo to upload from the current album name and the file name.

The function calls the `putObject` method of the Amazon S3 service object to upload the photo. After uploading the photo, the function redisplays the album so the uploaded photo appears.

```
// Add a photo to an album
const addPhoto = async (albumName) => {
  const files = document.getElementById("photoupload").files;
  try {
    const albumPhotosKey = encodeURIComponent(albumName) + "/";
    const data = await s3.send(
        new ListObjectsCommand({
          Prefix: albumPhotosKey,
          Bucket: albumBucketName
        })
    );
    const file = files[0];
    const fileName = file.name;
    const photoKey = albumPhotosKey + fileName;
    const uploadParams = {
      Bucket: albumBucketName,
      Key: photoKey,
      Body: file
    };
    try {
      const data = await s3.send(new PutObjectCommand(uploadParams));
      alert("Successfully uploaded photo.");
      viewAlbum(albumName);
    } catch (err) {
      return alert("There was an error uploading your photo: ", err.message);
    }
  } catch (err) {
    if (!files.length) {
      return alert("Choose a file to upload first.");
    }
  }
};
// Make addPhoto function available to the browser
window.addPhoto = addPhoto;
```

Deleting a photo

To delete a photo from an album in the Amazon S3 bucket, the application's `deletePhoto` function calls the `DeleteObjectCommand` method of the Amazon S3 client service object. This deletes the photo specified by the `photoKey` value passed to the function.

```
// Delete a photo from an album
const deletePhoto = async (albumName, photoKey) => {
  try {
    console.log(photoKey);
    const params = { Key: photoKey, Bucket: albumBucketName };
    const data = await s3.send(new DeleteObjectCommand(params));
    console.log("Successfully deleted photo.");
    viewAlbum(albumName);
  } catch (err) {
    return alert("There was an error deleting your photo: ", err.message);
```

```
  }
};
// Make deletePhoto function available to the browser
window.deletePhoto = deletePhoto;
```

Deleting an album

To delete an album in the Amazon S3 bucket, the application's `deleteAlbum` function calls the `deleteObjects` method of the Amazon S3 client service object.

```
// Delete an album from the bucket
const deleteAlbum = async (albumName) => {
  const albumKey = encodeURIComponent(albumName) + "/";
  try {
    const params = { Bucket: albumBucketName, Prefix: albumKey };
    const data = await s3.send(new ListObjectsCommand(params));
    const objects = data.Contents.map(function (object) {
      return { Key: object.Key };
    });
    try {
      const params = {
        Bucket: albumBucketName,
        Delete: { Objects: objects },
        Quiet: true,
      };
      const data = await s3.send(new DeleteObjectsCommand(params));
      listAlbums();
      return alert("Successfully deleted album.");
    } catch (err) {
      return alert("There was an error deleting your album: ", err.message);
    }
  } catch (err) {
    return alert("There was an error deleting your album1: ", err.message);
  }
};
// Make deleteAlbum function available to the browser
window.deleteAlbum = deleteAlbum;
```

Running the code

To run the code for this example

1. Save all the code as `s3_PhotoExample.js`.

   Note
   This file is available here on GitHub.

2. Replace *"REGION"* with your AWS Region, such as 'us-east-1'.
3. Replace *"BUCKET_NAME"* with your Amazon S3 bucket.
4. Replace *"IDENTITY_POOL_ID"* with the IdentityPoolId from the **Sample page** of the Amazon Cognito Identity Pool you created for this example.

   Note
   The *IDENTITY_POOL_ID* is displayed in red in the console.

5. Run the following in the command line to bundle the JavaScript for this example in to a file called `<main.js>`:

```
webpack s3_PhotoExample.ts --mode development --target web --devtool false -o main.js
```

Note
For information on installing WebPack, see Bundling applications with webpack (p. 42).

6.

Run the following in the command line:

```
node s3_PhotoUploader.ts
```

Uploading photos to Amazon S3: Full code

This section contains the full HTML and JavaScript code for the example in which photos are uploaded to an Amazon S3 photo album. See the parent section (p. 183) for details and prerequisites.

The HTML for the example:

```
<!DOCTYPE html>
<html>
<head>
    <script src="./main.js"></script>
    <script>
        function getHtml(template) {
            return template.join("\n");
        }
        listAlbums();
    </script>
</head>
<body>
<h1>My photo albums app</h1>
<div id="app"></div>
</body>
</html>
```

This sample code can be found here on GitHub.

The browser script code for the example:

```
// Load the required clients and packages
const { CognitoIdentityClient } = require("@aws-sdk/client-cognito-identity");
const {
  fromCognitoIdentityPool,
} = require("@aws-sdk/credential-provider-cognito-identity");
const { S3Client, PutObjectCommand, ListObjectsCommand, DeleteObjectCommand,
 DeleteObjectsCommand } = require("@aws-sdk/client-s3");

// Set the AWS Region
const REGION = "REGION"; //REGION

// Initialize the Amazon Cognito credentials provider
const s3 = new S3Client({
  region: REGION,
  credentials: fromCognitoIdentityPool({
    client: new CognitoIdentityClient({ region: REGION }),
    identityPoolId: "IDENTITY_POOL_ID", // IDENTITY_POOL_ID
  }),
});

const albumBucketName = "BUCKET_NAME"; //BUCKET_NAME

// A utility function to create HTML
```

```
function getHtml(template) {
  return template.join("\n");
}
// Make getHTML function available to the browser
window.getHTML = getHtml;

// List the photo albums that exist in the bucket
const listAlbums = async () => {
  try {
    const data = await s3.send(
        new ListObjectsCommand({ Delimiter: "/", Bucket: albumBucketName })
    );

    if (data.CommonPrefixes === undefined) {
      const htmlTemplate = [
        "<p>You don't have any albums. You need to create an album.</p>",
        "<button onclick=\"createAlbum(prompt('Enter album name:'))\">",
        "Create new album",
        "</button>",
      ];
      document.getElementById("app").innerHTML = htmlTemplate;
    } else {
      var albums = data.CommonPrefixes.map(function (commonPrefix) {
        var prefix = commonPrefix.Prefix;
        var albumName = decodeURIComponent(prefix.replace("/", ""));
        return getHtml([
          "<li>",
          "<span onclick=\"deleteAlbum('" + albumName + "')\">X</span>",
          "<span onclick=\"viewAlbum('" + albumName + "')\">",
          albumName,
          "</span>",
          "</li>",
        ]);
      });
      var message = albums.length
          ? getHtml([
            "<p>Click an album name to view it.</p>",
            "<p>Click the X to delete the album.</p>",
          ])
          : "<p>You do not have any albums. You need to create an album.";
      const htmlTemplate = [
        "<h2>Albums</h2>",
        message,
        "<ul>",
        getHtml(albums),
        "</ul>",
        "<button onclick=\"createAlbum(prompt('Enter Album Name:'))\">",
        "Create new Album",
        "</button>",
      ];
      document.getElementById("app").innerHTML = getHtml(htmlTemplate);
    }
  } catch (err) {
    return alert("There was an error listing your albums: " + err.message);
  }
};

// Make listAlbums function available to the browser
window.listAlbums = listAlbums;


// Create an album in the bucket
const createAlbum = async (albumName) => {
  albumName = albumName.trim();
  if (!albumName) {
    return alert("Album names must contain at least one non-space character.");
```

```
  }
  if (albumName.indexOf("/") !== -1) {
    return alert("Album names cannot contain slashes.");
  }
  var albumKey = encodeURIComponent(albumName);
  try {
    const key = albumKey + "/";
    const params = { Bucket: albumBucketName, Key: key };
    const data = await s3.send(new PutObjectCommand(params));
    alert("Successfully created album.");
    viewAlbum(albumName);
  } catch (err) {
    return alert("There was an error creating your album: " + err.message);
  }
};

// Make createAlbum function available to the browser
window.createAlbum = createAlbum;


// View the contents of an album

const viewAlbum = async (albumName) => {
  const albumPhotosKey = encodeURIComponent(albumName) + "/";
  try {
    const data = await s3.send(
        new ListObjectsCommand({
          Prefix: albumPhotosKey,
          Bucket: albumBucketName,
        })
    );
    if (data.Contents.length === 1) {
      var htmlTemplate = [
        "<p>You don't have any photos in this album. You need to add photos.</p>",
        '<input id="photoupload" type="file" accept="image/*">',
        '<button id="addphoto" onclick="addPhoto(\'' + albumName + "')\">",
        "Add photo",
        "</button>",
        '<button onclick="listAlbums()">',
        "Back to albums",
        "</button>",
      ];
      document.getElementById("app").innerHTML = getHtml(htmlTemplate);
    } else {
      console.log(data);
      const href = "https://s3." + REGION + ".amazonaws.com/";
      const bucketUrl = href + albumBucketName + "/";
      const photos = data.Contents.map(function (photo) {
        const photoKey = photo.Key;
        console.log(photo.Key);
        const photoUrl = bucketUrl + encodeURIComponent(photoKey);
        return getHtml([
          "<span>",
          "<div>",
          '<img style="width:128px;height:128px;" src="' + photoUrl + '"/>',
          "</div>",
          "<div>",
          "<span onclick=\"deletePhoto('" +
          albumName +
          "','" +
          photoKey +
          "')\">",
          "X",
          "</span>",
          "<span>",
          photoKey.replace(albumPhotosKey, ""),
```

```
          "</span>",
          "</div>",
          "</span>",
        ]);
      });
      var message = photos.length
          ? "<p>Click the X to delete the photo.</p>"
          : "<p>You don't have any photos in this album. You need to add photos.</p>";
      const htmlTemplate = [
        "<h2>",
        "Album: " + albumName,
        "</h2>",
        message,
        "<div>",
        getHtml(photos),
        "</div>",
        '<input id="photoupload" type="file" accept="image/*">',
        '<button id="addphoto" onclick="addPhoto(\'' + albumName + "')\">",
        "Add photo",
        "</button>",
        '<button onclick="listAlbums()">',
        "Back to albums",
        "</button>",
      ];
      document.getElementById("app").innerHTML = getHtml(htmlTemplate);
      document.getElementsByTagName("img")[0].remove();
    }
  } catch (err) {
    return alert("There was an error viewing your album: " + err.message);
  }
};
// Make viewAlbum function available to the browser
window.viewAlbum = viewAlbum;


// Add a photo to an album
const addPhoto = async (albumName) => {
  const files = document.getElementById("photoupload").files;
  try {
    const albumPhotosKey = encodeURIComponent(albumName) + "/";
    const data = await s3.send(
        new ListObjectsCommand({
          Prefix: albumPhotosKey,
          Bucket: albumBucketName
        })
    );
    const file = files[0];
    const fileName = file.name;
    const photoKey = albumPhotosKey + fileName;
    const uploadParams = {
      Bucket: albumBucketName,
      Key: photoKey,
      Body: file
    };
    try {
      const data = await s3.send(new PutObjectCommand(uploadParams));
      alert("Successfully uploaded photo.");
      viewAlbum(albumName);
    } catch (err) {
      return alert("There was an error uploading your photo: ", err.message);
    }
  } catch (err) {
    if (!files.length) {
      return alert("Choose a file to upload first.");
    }
  }
```

```
};
// Make addPhoto function available to the browser
window.addPhoto = addPhoto;


// Delete a photo from an album
const deletePhoto = async (albumName, photoKey) => {
  try {
    console.log(photoKey);
    const params = { Key: photoKey, Bucket: albumBucketName };
    const data = await s3.send(new DeleteObjectCommand(params));
    console.log("Successfully deleted photo.");
    viewAlbum(albumName);
  } catch (err) {
    return alert("There was an error deleting your photo: ", err.message);
  }
};
// Make deletePhoto function available to the browser
window.deletePhoto = deletePhoto;


// Delete an album from the bucket
const deleteAlbum = async (albumName) => {
  const albumKey = encodeURIComponent(albumName) + "/";
  try {
    const params = { Bucket: albumBucketName, Prefix: albumKey };
    const data = await s3.send(new ListObjectsCommand(params));
    const objects = data.Contents.map(function (object) {
      return { Key: object.Key };
    });
    try {
      const params = {
        Bucket: albumBucketName,
        Delete: { Objects: objects },
        Quiet: true,
      };
      const data = await s3.send(new DeleteObjectsCommand(params));
      listAlbums();
      return alert("Successfully deleted album.");
    } catch (err) {
      return alert("There was an error deleting your album: ", err.message);
    }
  } catch (err) {
    return alert("There was an error deleting your album1: ", err.message);
  }
};
// Make deleteAlbum function available to the browser
window.deleteAlbum = deleteAlbum;
```

This sample code can be found here on GitHub.

Amazon S3 Node.js examples

The following topics show examples of how the AWS SDK for JavaScript can be used to interact with Amazon S3 buckets using Node.js.

Topics

Creating and using Amazon S3 buckets

This Node.js code example shows:

- How to obtain and display a list of Amazon S3 buckets in your account.
- How to create an Amazon S3 bucket.
- How to upload an object to a specified bucket.

The scenario

In this example, a series of Node.js modules are used to obtain a list of existing Amazon S3 buckets, create a bucket, and upload a file to a specified bucket. These Node.js modules use the SDK for JavaScript to get information from and upload files to an Amazon S3 bucket using these methods of the Amazon S3 client class:

- `ListBucketsCommand`
- `CreateBucketCommand`
- `ListObjectsCommand`
- `PutObjectCommand`
- `UploadPartCommand`,
- `GetObjectCommand`
- `DeleteBucketCommand`

There is also an example that uses the following method of *node-fetch* to generate a presigned URL:

- `PUT`
- `GET`

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 Loading credentials in Node.js from the shared credentials file (p. 33).

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 (p. 56).

Displaying a list of Amazon S3 buckets

Create a `libs` directory, and create a Node.js module with the file name `s3Client.js`. Copy and paste the code below into it, which creates the Amazon S3 client object. Replace *REGION* with your AWS region.

```
import { S3Client} from "@aws-sdk/client-s3";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon S3 service client object.
const s3Client = new S3Client({ region: REGION });
export { s3Client };
```

This code is available here on GitHub.

Create a Node.js module with the file name `s3_listbuckets.js`. Make sure to configure the SDK as previously shown, including installing the required clients and packages. To access Amazon Simple Storage Service, create an `S3` client service object. Call the `listBuckets` method of the Amazon S3 client service object to retrieve a list of your buckets. The `data` parameter of the callback function has a `Buckets` property containing an array of maps to represent the buckets. Display the bucket list by logging it to the console.

```
// Import required AWS SDK clients and commands for Node.js
import { ListBucketsCommand } from "@aws-sdk/client-s3";
import { s3Client } from "./libs/s3Client.js"; // Helper function that creates Amazon S3
 service client module.

export const run = async () => {
  try {
    const data = await s3Client.send(new ListBucketsCommand({}));
    console.log("Success", data.Buckets);
    return data; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

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

```
node s3_listbuckets.js
```

This sample code can be found here on GitHub.

Creating an Amazon S3 bucket

Create a `libs` directory, and create a Node.js module with the file name `s3Client.js`. Copy and paste the code below into it, which creates the Amazon S3 client object. Replace *REGION* with your AWS region.

```
import { S3Client} from "@aws-sdk/client-s3";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon S3 service client object.
const s3Client = new S3Client({ region: REGION });
```

```
export { s3Client };
```

This code is available here on GitHub.

Create a Node.js module with the file name `s3_createbucket.js`. Make sure to configure the SDK as previously shown, including installing the required clients and packages. Create an `S3` client service object. The module will take a single command-line argument to specify a name for the new bucket.

Add a variable to hold the parameters used to call the `createBucket` method of the Amazon S3 client service object, including the name for the newly created bucket. The callback function logs the new bucket's location to the console after Amazon S3 successfully creates it.

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

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

```
node s3_createbucket.js
```

This sample code can be found here on GitHub.

Uploading a file to an Amazon S3 bucket

This section describes how to:

- Create a new object and upload it to an Amazon S3 bucket.
- Upload an existing object to an Amazon S3 bucket.

Create and upload an object to an Amazon S3 bucket

Create a `libs` directory, and create a Node.js module with the file name `s3Client.js`. Copy and paste the code below into it, which creates the Amazon S3 client object. Replace ***REGION*** with your AWS region.

```
import { S3Client} from "@aws-sdk/client-s3";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon S3 service client object.
const s3Client = new S3Client({ region: REGION });
export { s3Client };
```

This code is available here on GitHub.

Create a Node.js module with the file name `s3_create_and_upload_object.js`. Make sure to configure the SDK as previously shown, including installing the required clients and packages.

Create a variable with the parameters needed to call the `PutObjectCommand` method of the Amazon S3 service object. Provide the name of the target bucket in the `Bucket` parameter. For the `Key` parameter, provide a name for the object.

Note

To create a directory for the object, use the format `directoryY_NAME/OBJECT_NAME`.

```
// Import required AWS SDK clients and commands for Node.js
import { PutObjectCommand } from "@aws-sdk/client-s3";
import { s3Client } from "./libs/s3Client.js"; // Helper function that creates Amazon S3
 service client module.

// Set the parameters.
export const bucketParams = {
  Bucket: "BUCKET_NAME",
  // Specify the name of the new object. For example, 'index.html'.
  // To create a directory for the object, use '/'. For example, 'myApp/package.json'.
  Key: "OBJECT_NAME",
  // Content of the new object.
  Body: "BODY",
};

// Create and upload the object to the specified Amazon S3 bucket.
export const run = async () => {
  try {
    const data = await s3Client.send(new PutObjectCommand(bucketParams));
    return data; // For unit tests.
    console.log(
      "Successfully uploaded object: " +
        bucketParams.Bucket +
        "/" +
        bucketParams.Key
    );
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

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

```
node s3_create_and_upload_object.js
```

This sample code can be found here on GitHub.

Upload an existing object to an Amazon S3 bucket

Create a `libs` directory, and create a Node.js module with the file name `s3Client.js`. Copy and paste the code below into it, which creates the Amazon S3 client object. Replace *REGION* with your AWS region.

```
import { S3Client} from "@aws-sdk/client-s3";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon S3 service client object.
const s3Client = new S3Client({ region: REGION });
```

```
export { s3Client };
```

This code is available here on GitHub.

Create a Node.js module with the file name `s3_upload_object.js`. Make sure to configure the SDK as previously shown, including installing the required clients and packages.

Create a variable with the parameters needed to call the `PutObjectCommand` method of the Amazon S3 service object. Provide the name of the target bucket in the `Bucket` parameter. Provide the name of the existing object and the path to it. The `Key` parameter is set to the name of the selected file, which you can obtain using the Node.js `path` module.

```
// Import required AWS SDK clients and commands for Node.js.
import { PutObjectCommand } from "@aws-sdk/client-s3";
import { s3Client } from "./libs/s3Client.js"; // Helper function that creates Amazon S3
 service client module.
import {path} from "path";
import {fs} from "fs";

const file = "OBJECT_PATH_AND_NAME"; // Path to and name of object. For example '../
myFiles/index.js'.
const fileStream = fs.createReadStream(file);

// Set the parameters
export const uploadParams = {
  Bucket: "BUCKET_NAME",
  // Add the required 'Key' parameter using the 'path' module.
  Key: path.basename(file),
  // Add the required 'Body' parameter
  Body: fileStream,
};


// Upload file to specified bucket.
export const run = async () => {
  try {
    const data = await s3Client.send(new PutObjectCommand(uploadParams));
    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 s3_upload_object.js
```

This sample code can be found here on GitHub.

Getting a file from an Amazon S3 bucket

Create a `libs` directory, and create a Node.js module with the file name `s3Client.js`. Copy and paste the code below into it, which creates the Amazon S3 client object. Replace *`REGION`* with your AWS region.

```
import { S3Client} from "@aws-sdk/client-s3";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
```

```
// Create an Amazon S3 service client object.
const s3Client = new S3Client({ region: REGION });
export { s3Client };
```

This code is available here on GitHub.

Create a Node.js module with the file name `s3_getobject.js`. Make sure to configure the SDK as previously shown, including installing the required clients and packages. The module will take two command-line arguments, the first one to specify the target bucket and the second to specify the file to get.

Create a variable with the parameters needed to call the `GetObjectCommand` method of the Amazon S3 service object. Provide the name of the target bucket in the `Bucket` parameter. The `Key` parameter is set to the name of the file, which you can obtain using the Node.js `path` module.

```
// Import required AWS SDK clients and commands for Node.js.
import { GetObjectCommand } from "@aws-sdk/client-s3";
import { s3Client } from "./libs/s3Client.js"; // Helper function that creates Amazon S3
 service client module.

export const bucketParams = {
  Bucket: "BUCKET_NAME",
  Key: "KEY",
};

export const run = async () => {
  try {
    // Create a helper function to convert a ReadableStream to a string.
    const streamToString = (stream) =>
      new Promise((resolve, reject) => {
        const chunks = [];
        stream.on("data", (chunk) => chunks.push(chunk));
        stream.on("error", reject);
        stream.on("end", () => resolve(Buffer.concat(chunks).toString("utf8")));
      });

    // Get the object} from the Amazon S3 bucket. It is returned as a ReadableStream.
    const data = await s3Client.send(new GetObjectCommand(bucketParams));
      return data; // For unit tests.
    // Convert the ReadableStream to a string.
    const bodyContents = await streamToString(data.Body);
    console.log(bodyContents);
      return bodyContents;
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

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

```
node s3_upload.js
```

This sample code can be found here on GitHub.

Listing objects in an Amazon S3 bucket

This example lists up to 1000 objects in an Amazon S3 Bucket.

> **Note**
> To list more than 1000 objects, see Listing more than 1000 objects in an Amazon S3 bucket (p. 204).

Create a `libs` directory, and create a Node.js module with the file name `s3Client.js`. Copy and paste the code below into it, which creates the Amazon S3 client object. Replace *REGION* with your AWS region.

```
import { S3Client} from "@aws-sdk/client-s3";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon S3 service client object.
const s3Client = new S3Client({ region: REGION });
export { s3Client };
```

This code is available here on GitHub.

Create a Node.js module with the file name `s3_listobjects.js`. Make sure to configure the SDK as previously shown, including installing the required clients and packages.

Add a variable to hold the parameters used to call the `ListObjectsCommnad` method of the Amazon S3 service object, including the name of the bucket to read. The callback function logs a list of objects (files) or a failure message.

```
// Import required AWS SDK clients and commands for Node.js
import { ListObjectsCommand } from "@aws-sdk/client-s3";
import { s3Client } from "./libs/s3Client.js"; // Helper function that creates Amazon S3
 service client module.

// Create the parameters for the bucket
export const bucketParams = { Bucket: "BUCKET_NAME" };

export const run = async () => {
  try {
    const data = await s3Client.send(new ListObjectsCommand(bucketParams));
    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 s3_listobjects.js
```

This sample code can be found here on GitHub.

Listing more than 1000 objects in an Amazon S3 bucket

This example lists more than 1000 objects in an Amazon S3 Bucket.

Create a `libs` directory, and create a Node.js module with the file name `s3Client.js`. Copy and paste the code below into it, which creates the Amazon S3 client object. Replace *REGION* with your AWS region.

```
import { S3Client} from "@aws-sdk/client-s3";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon S3 service client object.
const s3Client = new S3Client({ region: REGION });
export { s3Client };
```

This code is available here on GitHub.

Create a Node.js module with the file name `s3_list1000plusobjects.js`. Make sure to configure the SDK as previously shown, including installing the required clients and packages. Create an `S3` client service object.

Use a while loop to list each 1000 items until all items have been listed. Then declare `truncated` as a flag with a value of `true`, and a while loop that prints 1,000 items at at time, until the the flag is `false`.

```
// Import required AWS SDK clients and commands for Node.js
import { ListObjectsCommand } from "@aws-sdk/client-s3";
import { s3Client } from "./libs/s3Client.js"; // Helper function that creates Amazon S3
 service client module.

// Create the parameters for the bucket
export const bucketParams = { Bucket: "BUCKET_NAME" };

export async function run() {
  // Declare truncated as a flag that we will base our while loop on
  let truncated = true;
  // Declare a variable that we will assign the key of the last element in the response to
  let pageMarker;
  // While loop that runs until response.truncated is false
  while (truncated) {
    try {
      const response = await s3Client.send(new ListObjectsCommand(bucketParams));
      // return response; //For unit tests
      response.Contents.forEach((item) => {
        console.log(item.Key);
      });
      // Log the Key of every item in the response to standard output
      truncated = response.IsTruncated;
      // If 'truncated' is true, assign the key of the final element in the response to our
 variable 'pageMarker'
      if (truncated) {
        pageMarker = response.Contents.slice(-1)[0].Key;
        // Assign value of pageMarker to bucketParams so that the next iteration will
 start} from the new pageMarker.
        bucketParams.Marker = pageMarker;
      }
      // At end of the list, response.truncated is false and our function exits the while
 loop.
    } catch (err) {
      console.log("Error", err);
      truncated = false;
    }
  }
}
run();
```

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

```
node s3_list1000plusobjects.js
```

This sample code can be found here on GitHub.

Deleting an Amazon S3 bucket

Create a `libs` directory, and create a Node.js module with the file name `s3Client.js`. Copy and paste the code below into it, which creates the Amazon S3 client object. Replace *`REGION`* with your AWS region.

```
import { S3Client} from "@aws-sdk/client-s3";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon S3 service client object.
const s3Client = new S3Client({ region: REGION });
export { s3Client };
```

This code is available here on GitHub.

Create a Node.js module with the file name `s3_deletebucket.js`. Make sure to configure the SDK as previously shown, including installing the required clients and packages.

Add a variable to hold the parameters used to call the `deleteBucket` method of the Amazon S3 service object, including the name of the bucket to delete. The bucket must be empty to delete it. The callback function logs a success or failure message.

```
// Import required AWS SDK clients and commands for Node.js
import { DeleteBucketCommand } from "@aws-sdk/client-s3";
import { s3Client } from "./libs/s3Client.js"; // Helper function that creates Amazon S3
 service client module.

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

export const run = async () => {
  try {
    const data = await s3Client.send(new DeleteBucketCommand(bucketParams));
    return data; // For unit tests.
    console.log("Success - bucket deleted");
  } catch (err) {
    console.log("Error", err);
  }
};
// Invoke run() so these examples run out of the box.
run();
```

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

```
node s3_deletebucket.js
```

This sample code can be found here on GitHub.

Creating a presigned URL

This section demonstrated how to create presigned URLs to get and put objects in Amazon S3 buckets.

Create a presigned URL to upload objects to an Amazon S3 bucket

Create a `libs` directory, and create a Node.js module with the file name `s3Client.js`. Copy and paste the code below into it, which creates the Amazon S3 client object. Replace ***REGION*** with your AWS region.

```
import { S3Client} from "@aws-sdk/client-s3";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon S3 service client object.
const s3Client = new S3Client({ region: REGION });
export { s3Client };
```

This code is available here on GitHub.

Create a Node.js module with the file name `s3_presignedURL_v3.js`. Make sure to configure the SDK as previously shown, including installing the required clients and packages.

Create a function that creates a bucket and an object to upload, creates a presigned url to upload the object, and then uploads the object. In this example, the object and bucket are automatically deleted to ensure you don't incur any unnecessary expense.

Create a variable with the parameters needed to call the `PutObjectCommand` command of the Amazon S3 service object. The bucket name, filename, or key, body, and duration to expiration are prepopulated in this example.

For more information on creating presigned URLs, see https://docs.aws.amazon.com/AmazonS3/latest/dev/PresignedUrlUploadObject.html.

```
// Import the required AWS SDK clients and commands for Node.js
import {
  CreateBucketCommand,
  DeleteObjectCommand,
  PutObjectCommand,
  DeleteBucketCommand }
from "@aws-sdk/client-s3";
import { s3Client } from "./libs/s3Client.js"; // Helper function that creates Amazon S3
 service client module.
import { getSignedUrl } from "@aws-sdk/s3-request-presigner";
import fetch from "node-fetch";

// Set parameters
// Create a random names for the Amazon Simple Storage Service (Amazon S3) bucket and key
export const bucketParams = {
  Bucket: `test-bucket-${Math.ceil(Math.random() * 10 ** 10)}`,
  Key: `test-object-${Math.ceil(Math.random() * 10 ** 10)}`,
  Body: "BODY"
};
export const run = async () => {
  try {
    // Create an Amazon S3 bucket.
    console.log(`Creating bucket ${bucketParams.Bucket}`);
    await s3Client.send(new CreateBucketCommand({ Bucket: bucketParams.Bucket }));
    console.log(`Waiting for "${bucketParams.Bucket}" bucket creation...`);
  } catch (err) {
    console.log("Error creating bucket", err);
  }
  try {
    // Create the command.
    const command = new PutObjectCommand(bucketParams);

    // Create the presigned URL.
    const signedUrl = await getSignedUrl(s3Client, command, {
      expiresIn: 3600,
    });
    console.log(
      `\nPutting "${bucketParams.Key}" using signedUrl with body "${bucketParams.Body}" in
 v3`
    );
    console.log(signedUrl);
    const response = await fetch(signedUrl);
    console.log(
      `\nResponse returned by signed URL: ${await response.text()}\n`
    );
    return response;
  } catch (err) {
    console.log("Error creating presigned URL", err);
```

```
  }
  try {
    // Delete the object.
    console.log(`\nDeleting object "${bucketParams.Key}"} from bucket`);
    await s3Client.send(
      new DeleteObjectCommand({ Bucket: bucketParams.Bucket, Key: bucketParams.Key })
    );
  } catch (err) {
    console.log("Error deleting object", err);
  }
  try {
    // Delete the Amazon S3 bucket.
    console.log(`\nDeleting bucket ${bucketParams.Bucket}`);
    await s3.send(new DeleteBucketCommand({ Bucket: bucketParams.Bucket }));
  } catch (err) {
    console.log("Error deleting bucket", err);
  }
};
run();
```

To run the example, type the following at the command line.

```
node s3_put_presignedURL_v3.js
```

This sample code can be found here on GitHub.

Create a presigned URL to get objects from an Amazon S3 bucket

Create a `libs` directory, and create a Node.js module with the file name `s3Client.js`. Copy and paste the code below into it, which creates the Amazon S3 client object. Replace *REGION* with your AWS region.

```
import { S3Client} from "@aws-sdk/client-s3";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon S3 service client object.
const s3Client = new S3Client({ region: REGION });
export { s3Client };
```

This code is available here on GitHub.

Create a Node.js module with the file name `s3_get_presignedURL_v3.js`. Make sure to configure the SDK as previously shown, including installing the required clients and packages.

Create a function that creates a bucket and uploads, and object, then creates a presigned url to get the object from the bucket. In this example, the object and bucket are automatically deleted to ensure you don't incur any unnecessary expense.

Create a variable with the parameters needed to call the `GetObjectCommand` command of the Amazon S3 service object. The bucket name, filename, or key, body, and duration to expiration are prepopulated in this example.

For more information on creating presigned URLs, see https://docs.aws.amazon.com/AmazonS3/latest/dev/PresignedUrlUploadObject.html.

```
// Import the required AWS SDK clients and commands for Node.js
import {
  CreateBucketCommand,
  PutObjectCommand,
```

```
  GetObjectCommand,
  DeleteObjectCommand,
  DeleteBucketCommand }
from "@aws-sdk/client-s3";
import { s3Client } from "./libs/s3Client.js"; // Helper function that creates Amazon S3
 service client module.
import { getSignedUrl } from "@aws-sdk/s3-request-presigner";
const fetch = require("node-fetch");

// Set parameters
// Create random names for the Amazon Simple Storage Service (Amazon S3) bucket and key.
export const bucketParams = {
  Bucket: `test-bucket-${Math.ceil(Math.random() * 10 ** 10)}`,
  Key: `test-object-${Math.ceil(Math.random() * 10 ** 10)}`,
  Body: "BODY"
};

export const run = async () => {
  // Create an Amazon S3 bucket.
  try {
    console.log(`Creating bucket ${bucketParams.Bucket}`);
    const data = await s3Client.send(
      new CreateBucketCommand({ Bucket: bucketParams.Bucket })
    );
    return data; // For unit tests.
    console.log(`Waiting for "${bucketParams.Bucket}" bucket creation...\n`);
  } catch (err) {
    console.log("Error creating bucket", err);
  }
  // Put the object in the Amazon S3 bucket.
  try {
    console.log(`Putting object "${bucketParams.Key}" in bucket`);
    const data = await s3Client.send(
      new PutObjectCommand({
        Bucket: bucketParams.Bucket,
        Key: bucketParams.Key,
        Body: bucketParams.Body,
      })
    );
    return data; // For unit tests.
  } catch (err) {
    console.log("Error putting object", err);
  }
  // Create a presigned URL.
  try {
    // Create the command.
    const command = new GetObjectCommand(bucketParams);

    // Create the presigned URL.
    const signedUrl = await getSignedUrl(s3Client, command, {
      expiresIn: 3600,
    });
    console.log(
      `\nGetting "${bucketParams.Key}" using signedUrl with body "${bucketParams.Body}" in
 v3`
    );
    console.log(signedUrl);
    const response = await fetch(signedUrl);
    console.log(
      `\nResponse returned by signed URL: ${await response.text()}\n`
    );
  } catch (err) {
    console.log("Error creating presigned URL", err);
  }
  // Delete the object.
  try {
```

```
    console.log(`\nDeleting object "${bucketParams.Key}"} from bucket`);
    const data = await s3Client.send(
      new DeleteObjectCommand({ Bucket: bucketParams.Bucket, Key: bucketParams.Key })
    );
    return data; // For unit tests.
  } catch (err) {
    console.log("Error deleting object", err);
  }
  // Delete the bucket.
  try {
    console.log(`\nDeleting bucket ${bucketParams.Bucket}`);
    const data = await s3Client.send(
      new DeleteBucketCommand({ Bucket: bucketParams.Bucket, Key: bucketParams.Key })
    );
    return data; // For unit tests.
  } catch (err) {
    console.log("Error deleting object", err);
  }
};
run();
```

To run the example, type the following at the command line.

```
node s3_get_presignedURL_v3.js
```

This sample code can be found here on GitHub.

Configuring Amazon S3 buckets

This Node.js code example shows:

- How to configure the cross-origin resource sharing (CORS) permissions for a bucket.

The scenario

In this example, a series of Node.js modules are used to list your Amazon S3 buckets and to configure CORS and bucket logging. The Node.js modules use the SDK for JavaScript to configure a selected Amazon S3 bucket using these methods of the Amazon S3 client class:

- `GetBucketCorsCommand`
- `PutBucketCorsCommand`

For more information about using CORS configuration with an Amazon S3 bucket, see Cross-origin resource sharing (CORS) in the *Amazon Simple Storage Service User Guide*.

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 Loading credentials in Node.js from the shared credentials file (p. 33).

> **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 (p. 56).

Retrieving a bucket CORS configuration

Create a `libs` directory, and create a Node.js module with the file name `s3Client.js`. Copy and paste the code below into it, which creates the Amazon S3 client object. Replace *`REGION`* with your AWS region.

```
import { S3Client} from "@aws-sdk/client-s3";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon S3 service client object.
const s3Client = new S3Client({ region: REGION });
export { s3Client };
```

Create a Node.js module with the file name `s3_getcors.js`. The module will take a single command-line argument to specify the bucket whose CORS configuration you want. Make sure to configure the SDK as previously shown, including installing the required clients and packages. Create an `S3` client service object.

The only parameter you need to pass is the name of the selected bucket when calling the `GetBucketCorsCommand` method. If the bucket currently has a CORS configuration, that configuration is returned by Amazon S3 as the `CORSRules` property of the `data` parameter passed to the callback function.

If the selected bucket has no CORS configuration, that information is returned to the callback function in the `error` parameter.

```

// Import required AWS SDK clients and commands for Node.js
import { GetBucketCorsCommand } from "@aws-sdk/client-s3";
import { s3Client } from "./libs/s3Client.js"; // Helper function that creates Amazon S3
 service client module.

// Create the parameters for calling
export const bucketParams = { Bucket: "BUCKET_NAME" };

export const run = async () => {
  try {
    const data = await s3Client.send(new GetBucketCorsCommand(bucketParams));
    console.log("Success", JSON.stringify(data.CORSRules));
    return data; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

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

```
node s3_getcors.js
```

This sample code can be found here on GitHub.

Setting a bucket CORS configuration

Create a `libs` directory, and create a Node.js module with the file name `s3Client.js`. Copy and paste the code below into it, which creates the Amazon S3 client object. Replace *REGION* with your AWS region.

```
import { S3Client} from "@aws-sdk/client-s3";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon S3 service client object.
const s3Client = new S3Client({ region: REGION });
export { s3Client };
```

This code is available here on GitHub.

Create a Node.js module with the file name `s3_setcors.js`. The module takes multiple command-line arguments, the first of which specifies the bucket whose CORS configuration you want to set. Additional arguments enumerate the HTTP methods (POST, GET, PUT, PATCH, DELETE, POST) you want to allow for the bucket. Configure the SDK as previously shown, including installing the required clients and packages.

Next create a JSON object to hold the values for the CORS configuration as required by the `PutBucketCorsCommand` method of the `S3` service object. Specify `"Authorization"` for the `AllowedHeaders` value and `"*"` for the `AllowedOrigins` value. Set the value of `AllowedMethods` as empty array initially.

Specify the allowed methods as command line parameters to the Node.js module, adding each of the methods that match one of the parameters. Add the resulting CORS configuration to the array of configurations contained in the `CORSRules` parameter. Specify the bucket you want to configure for CORS in the `Bucket` parameter.

```
 // Import required AWS-SDK clients and commands for Node.js
import { PutBucketCorsCommand } from "@aws-sdk/client-s3";
import { s3Client } from "./libs/s3Client.js"; // Helper function that creates Amazon S3
service client module.

 // Set params
 // Create initial parameters JSON for putBucketCors
 const thisConfig = {
   AllowedHeaders: ["Authorization"],
   AllowedMethods: [],
   AllowedOrigins: ["*"],
   ExposeHeaders: [],
   MaxAgeSeconds: 3000,
 };

 // Assemble the list of allowed methods based on command line parameters
 const allowedMethods = [];
 process.argv.forEach(function (val, index, array) {
   if (val.toUpperCase() === "POST") {
     allowedMethods.push("POST");
   }
   if (val.toUpperCase() === "GET") {
     allowedMethods.push("GET");
   }
   if (val.toUpperCase() === "PUT") {
```

```
      allowedMethods.push("PUT");
    }
    if (val.toUpperCase() === "PATCH") {
      allowedMethods.push("PATCH");
    }
    if (val.toUpperCase() === "DELETE") {
      allowedMethods.push("DELETE");
    }
    if (val.toUpperCase() === "HEAD") {
      allowedMethods.push("HEAD");
    }
  });

  // Copy the array of allowed methods into the config object
  thisConfig.AllowedMethods = allowedMethods;

  // Create array of configs then add the config object to it
  const corsRules = new Array(thisConfig);

  // Create CORS params
export  const corsParams = {
    Bucket: "BUCKET_NAME",
    CORSConfiguration: { CORSRules: corsRules },
  };
export async function run() {
  try {
    const data = await s3Client.send(new PutBucketCorsCommand(corsParams));
    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 including one or more HTTP methods as shown.

```
node s3_setcors.js
```

This sample code can be found here on GitHub.

Managing Amazon S3 bucket access permissions

This Node.js code example shows:

- How to retrieve or set the access control list for an Amazon S3 bucket.

The scenario

In this example, a Node.js module is used to display the bucket access control list (ACL) for a selected bucket and apply changes to the ACL for a selected bucket. The Node.js module uses the SDK for JavaScript to manage Amazon S3 bucket access permissions using these methods of the Amazon S3 client class:

- `GetBucketAclCommand`
- `PutBucketAclCommand`

For more information about access control lists for Amazon S3 buckets, see Managing access with ACLs in the *Amazon Simple Storage Service User Guide*.

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 Loading credentials in Node.js from the shared credentials file (p. 33).

> **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 (p. 56).

Retrieving the current bucket Access Control List

Create a `libs` directory, and create a Node.js module with the file name `s3Client.js`. Copy and paste the code below into it, which creates the Amazon S3 client object. Replace *`REGION`* with your AWS region.

```
import { S3Client} from "@aws-sdk/client-s3";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon S3 service client object.
const s3Client = new S3Client({ region: REGION });
export { s3Client };
```

This code is available here on GitHub.

Create a Node.js module with the file name `s3_getbucketacl.js`. Make sure to configure the SDK as previously shown, including installing the required clients and packages.

Create an `S3Client` client service object. The only parameter you need to pass is the name of the selected bucket when calling the `GetBucketAclCommand` method. The current access control list configuration is returned by Amazon S3 in the `data` parameter passed to the callback function.

```
// Import required AWS SDK clients and commands for Node.js
import { GetBucketAclCommand } from "@aws-sdk/client-s3";
import { s3Client } from "./libs/s3Client.js"; // Helper function that creates Amazon S3
 service client module.

// Create the parameters.
export const bucketParams = { Bucket: "BUCKET_NAME" };

export const run = async () => {
  try {
    const data = await s3Client.send(new GetBucketAclCommand(bucketParams));
```

```
    console.log("Success", data.Grants);
    return data; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

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

```
node s3_getbucketacl.js
```

This sample code can be found here on GitHub.

Attaching Access Control List permissions to an Amazon S3 bucket

Create a `libs` directory, and create a Node.js module with the file name `s3Client.js`. Copy and paste the code below into it, which creates the Amazon S3 client object. Replace *REGION* with your AWS region.

```
import { S3Client} from "@aws-sdk/client-s3";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon S3 service client object.
const s3Client = new S3Client({ region: REGION });
export { s3Client };
```

This code is available here on GitHub.

Create a Node.js module with the file name `s3_putbucketacl.js`. Make sure to configure the SDK as previously shown, including installing the required clients and packages.

Replace *BUCKET_NAME* with the name of the Amazon S3 bucket. Replace *GRANTEE_1* and *GRANTEE_2* with users you want to grant respective access contol permission.

```
/ Import required AWS SDK; clients and commands for Node.js.
const { S3Client, PutBucketAclCommand } = require("@aws-sdk/client-s3");

// Set the parameters.
const bucketParams = {
    Bucket: "BUCKET_NAME",
    // 'GrantFullControl' allows grantee the read, write, read ACP, and write ACP
 permissions on the bucket.
    // For example, an AWS account Canonical User ID in the format:
    // id=002160194XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXa7a49125274
    GrantFullControl:
        "GRANTEE_1",
    // 'GrantWrite' allows grantee to create, overwrite, and delete any object in the
 bucket..
    // For example, 'uri=http://acs.amazonaws.com/groups/s3/LogDelivery'
    GrantWrite: "GRANTEE_2"
};

// Create an Amazon S3 client service object.
const s3 = new S3Client({});

const run = async () => {
    try {
        const data = await s3.send(new PutBucketAclCommand(bucketParams));
        console.log("Success, permissions added to bucket", data);
    } catch (err) {
```

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

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

```
node s3_putbucketacl.js
```

This sample code can be found here on GitHub.

Working with Amazon S3 bucket policies

This Node.js code example shows:

- How to retrieve the bucket policy of an Amazon S3 bucket.
- How to add or update the bucket policy of an Amazon S3 bucket.
- How to delete the bucket policy of an Amazon S3 bucket.

The scenario

In this example, a series of Node.js modules are used to retrieve, set, or delete a bucket policy on an Amazon S3 bucket. The Node.js modules use the SDK for JavaScript to configure policy for a selected Amazon S3 bucket using these methods of the Amazon S3 client class:

- `GetBucketPolicyCommand`
- `PutBucketPolicyCommand`
- `DeleteBucketPolicyCommand`

For more information about bucket policies for Amazon S3 buckets, see Using bucket policies and user policies in the *Amazon Simple Storage Service User Guide*.

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 Loading credentials in Node.js from the shared credentials file (p. 33).

> **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 (p. 56).

Retrieving the current bucket policy

Create a `libs` directory, and create a Node.js module with the file name `s3Client.js`. Copy and paste the code below into it, which creates the Amazon S3 client object. Replace *REGION* with your AWS region.

```
import { S3Client} from "@aws-sdk/client-s3";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon S3 service client object.
const s3Client = new S3Client({ region: REGION });
export { s3Client };
```

This code is available here on GitHub.

Create a Node.js module with the file name `s3_getbucketpolicy.js`. The module takes a single command-line argument that specifies the bucket whose policy you want. Make sure to configure the SDK as previously shown, including installing the required clients and packages.

Create an `S3` service object. The only parameter you need to pass is the name of the selected bucket when calling the `GetBucketPolicyCommand` method. If the bucket currently has a policy, that policy is returned by Amazon S3 in the `data` parameter passed to the callback function.

If the selected bucket has no policy, that information is returned to the callback function in the `error` parameter.

```
// Import required AWS SDK clients and commands for Node.js
import { GetBucketPolicyCommand } from "@aws-sdk/client-s3";
import { s3Client } from "./libs/s3Client.js"; // Helper function that creates Amazon S3
 service client module.

// Create the parameters for calling
export const bucketParams = { Bucket: "BUCKET_NAME" };

export const run = async () => {
  try {
    const data = await s3Client.send(new GetBucketPolicyCommand(bucketParams));
    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 s3_getbucketpolicy.js
```

This sample code can be found here on GitHub.

Setting a simple bucket policy

Create a `libs` directory, and create a Node.js module with the file name `s3Client.js`. Copy and paste the code below into it, which creates the Amazon S3 client object. Replace *REGION* with your AWS region.

```
import { S3Client} from "@aws-sdk/client-s3";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon S3 service client object.
const s3Client = new S3Client({ region: REGION });
export { s3Client };
```

This code is available here on GitHub.

Create a Node.js module with the file name `s3_setbucketpolicy.js`. The module takes a single command-line argument that specifies the bucket whose policy you want to apply. Configure the SDK as previously shown, including installing the required clients and packages.

Bucket policies are specified in JSON. First, create a JSON object that contains all of the values to specify the policy except for the `Resource` value that identifies the bucket.

Format the `Resource` string required by the policy, incorporating the name of the selected bucket. Insert that string into the JSON object. Prepare the parameters for the `PutBucketPolicyCommand` method, including the name of the bucket and the JSON policy converted to a string value.

```
// Import required AWS SDK clients and commands for Node.js
import { PutBucketPolicyCommand } from "@aws-sdk/client-s3";
import { s3Client } from "./libs/s3Client.js"; // Helper function that creates Amazon S3
 service client module.

// Create params JSON for S3.createBucket
const BUCKET_NAME = "BUCKET_NAME";
export const bucketParams = {
  Bucket: BUCKET_NAME,
};
// Create the policy
const readOnlyAnonUserPolicy = {
  Version: "2012-10-17",
  Statement: [
    {
      Sid: "AddPerm",
      Effect: "Allow",
      Principal: "*",
      Action: ["s3:GetObject"],
      Resource: [""],
    },
  ],
};

// create selected bucket resource string for bucket policy
const bucketResource = "arn:aws:s3:::" + BUCKET_NAME + "/*"; //BUCKET_NAME
readOnlyAnonUserPolicy.Statement[0].Resource[0] = bucketResource;

// // convert policy JSON into string and assign into params
const bucketPolicyParams = {
  Bucket: BUCKET_NAME,
  Policy: JSON.stringify(readOnlyAnonUserPolicy),
};

export const run = async () => {
  try {
    // const response = await s3.putBucketPolicy(bucketPolicyParams);
    const response = await s3Client.send(
      new PutBucketPolicyCommand(bucketPolicyParams)
    );
    return response;
    console.log("Success, permissions added to bucket", response);
  } catch (err) {
```

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

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

```
node s3_setbucketpolicy.js
```

This sample code can be found here on GitHub.

Deleting a bucket policy

Create a `libs` directory, and create a Node.js module with the file name `s3Client.js`. Copy and paste the code below into it, which creates the Amazon S3 client object. Replace *`REGION`* with your AWS region.

```
import { S3Client} from "@aws-sdk/client-s3";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon S3 service client object.
const s3Client = new S3Client({ region: REGION });
export { s3Client };
```

This code is available here on GitHub.

Create a Node.js module with the file name `s3_deletebucketpolicy.js`. The module takes a single command-line argument that specifies the bucket whose policy you want to delete. Configure the SDK as previously shown, including installing the required clients and packages.

The only parameter you need to pass when calling the `DeleteBucketPolicy` method is the name of the selected bucket.

```

// Import required AWS SDK clients and commands for Node.js
import { DeleteBucketPolicyCommand } from "@aws-sdk/client-s3";
import { s3Client } from "./libs/s3Client.js"; // Helper function that creates Amazon S3
 service client module.

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

export const run = async () => {
  try {
    const data = await s3Client.send(new DeleteBucketPolicyCommand(bucketParams));
    console.log("Success", data + ", bucket policy deleted");
    return data; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
// Invoke run() so these examples run out of the box.
run();
```

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

```
node s3_deletebucketpolicy.js
```

This sample code can be found here on GitHub.

Using an Amazon S3 bucket as a static web host

This Node.js code example shows:

- How to set up an Amazon S3 bucket as a static web host.

The scenario

In this example, a series of Node.js modules are used to configure any of your buckets to act as a static web host. The Node.js modules use the SDK for JavaScript to configure a selected Amazon S3 bucket using these methods of the Amazon S3 client class:

- `GetBucketWebsiteCommand`
- `PutBucketWebsiteCommand`
- `DeleteBucketWebsiteCommand`

For more information about using an Amazon S3 bucket as a static web host, see Hosting a static website on Amazon S3 in the *Amazon Simple Storage Service User Guide*.

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 Loading credentials in Node.js from the shared credentials file (p. 33).

> **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 (p. 56).

Retrieving the current bucket website configuration

Create a `libs` directory, and create a Node.js module with the file name `s3Client.js`. Copy and paste the code below into it, which creates the Amazon S3 client object. Replace *`REGION`* with your AWS region.

```
import { S3Client} from "@aws-sdk/client-s3";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
```

```
// Create an Amazon S3 service client object.
const s3Client = new S3Client({ region: REGION });
export { s3Client };
```

This code is available here on GitHub.

Create a Node.js module with the file name `s3_getbucketwebsite.js`. The module takes a single command-line argument that specifies the bucket whose website configuration you want. Configure the SDK as previously shown, including installing the required clients and packages.

Create a function that retrieves the current bucket website configuration for the bucket selected in the bucket list. The only parameter you need to pass is the name of the selected bucket when calling the `GetBucketWebsiteCommand` method. If the bucket currently has a website configuration, that configuration is returned by Amazon S3 in the `data` parameter passed to the callback function.

If the selected bucket has no website configuration, that information is returned to the callback function in the `err` parameter.

```
// Import required AWS SDK clients and commands for Node.js
import { GetBucketWebsiteCommand } from "@aws-sdk/client-s3";
import { s3Client } from "./libs/s3Client.js"; // Helper function that creates Amazon S3
 service client module.

// Create the parameters for calling
export const bucketParams = { Bucket: "BUCKET_NAME" };

export const run = async () => {
  try {
    const data = await s3Client.send(new GetBucketWebsiteCommand(bucketParams));
    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 s3_getbucketwebsite.js
```

This sample code can be found here on GitHub.

Setting a bucket website configuration

Create a `libs` directory, and create a Node.js module with the file name `s3Client.js`. Copy and paste the code below into it, which creates the Amazon S3 client object. Replace *`REGION`* with your AWS region.

```
import { S3Client} from "@aws-sdk/client-s3";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon S3 service client object.
const s3Client = new S3Client({ region: REGION });
export { s3Client };
```

This code is available here on GitHub.

Create a Node.js module with the file name `s3_setbucketwebsite.js`. Make sure to configure the SDK as previously shown, including installing the required clients and packages. Create an `S3` client service object.

Create a function that applies a bucket website configuration. The configuration allows the selected bucket to serve as a static web host. Website configurations are specified in JSON. First, create a JSON object that contains all the values to specify the website configuration, except for the `Key` value that identifies the error document, and the `Suffix` value that identifies the index document.

Insert the values of the text input elements into the JSON object. Prepare the parameters for the `PutBucketWebsiteCommand` method, including the name of the bucket and the JSON website configuration.

```
// Import required AWS SDK clients and commands for Node.js
import { PutBucketWebsiteCommand } from "@aws-sdk/client-s3";
import { s3Client } from "./libs/s3Client.js"; // Helper function that creates Amazon S3
 service client module.

// Create the parameters for the bucket
export const bucketParams = { Bucket: "BUCKET_NAME" };
export const staticHostParams = {
  Bucket: bucketParams,
  WebsiteConfiguration: {
    ErrorDocument: {
      Key: "",
    },
    IndexDocument: {
      Suffix: "",
    },
  },
};

export const run = async () => {
  // Insert specified bucket name and index and error documents into params JSON
  //} from command line arguments
  staticHostParams.Bucket = bucketParams;
  staticHostParams.WebsiteConfiguration.IndexDocument.Suffix = "INDEX_PAGE"; // the index
 document inserted into params JSON
  staticHostParams.WebsiteConfiguration.ErrorDocument.Key = "ERROR_PAGE"; // : the error
 document inserted into params JSON
  // set the new website configuration on the selected bucket
  try {
    const data = await s3Client.send(new PutBucketWebsiteCommand(staticHostParams));
    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 s3_setbucketwebsite.js
```

This sample code can be found here on GitHub.

Deleting a bucket website configuration

Create a `libs` directory, and create a Node.js module with the file name `s3Client.js`. Copy and paste the code below into it, which creates the Amazon S3 client object. Replace *REGION* with your AWS region.

```
import { S3Client} from "@aws-sdk/client-s3";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create an Amazon S3 service client object.
const s3Client = new S3Client({ region: REGION });
export { s3Client };
```

This code is available here on GitHub.

Create a Node.js module with the file name `s3_deletebucketwebsite.js`. Make sure to configure the SDK as previously shown, including installing the required clients and packages. Create an `S3` client service object.

Create a function that deletes the website configuration for the selected bucket. The only parameter you need to pass when calling the `DeleteBucketWebsiteCommand` method is the name of the selected bucket.

```

// Import required AWS SDK clients and commands for Node.js

import { DeleteBucketWebsiteCommand } from "@aws-sdk/client-s3";
import { s3Client } from "./libs/s3Client.js"; // Helper function that creates Amazon S3
 service client module.

// Create the parameters for calling
export const bucketParams = { Bucket: "BUCKET_NAME" };

export const run = async () => {
  try {
    const data = await s3Client.send(new DeleteBucketWebsiteCommand(bucketParams));
    return data; // For unit tests.
    console.log("Success", data);
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

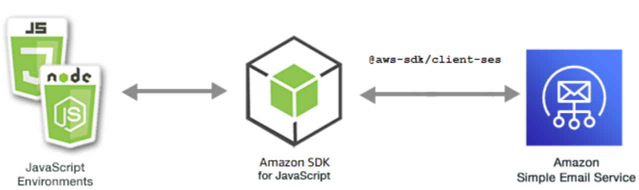
To run the example, enter the following at the command prompt.

```
node s3_deletebucketwebsite.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

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 credentials JSON file, see Loading credentials in Node.js from the shared credentials file (p. 33).

> **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 (p. 56).

Listing your identities

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

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

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

```
// Import required AWS SDK clients and commands for Node.js
import { ListIdentitiesCommand }  from "@aws-sdk/client-ses";
import { sesClient } from "./libs/sesClient.js";
// Set the parameters
var params = {
  IdentityType: "EmailAddress", // IDENTITY_TYPE: "EmailAddress' or 'Domain'
  MaxItems: 10,
};

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

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

```
node ses_listidentities.js
```

This example code can be found here on GitHub.

Verifying an email address identity

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

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

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

This example code can be found here on GitHub.

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

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

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

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

// Set the parameters
const params = { EmailAddress: "ADDRESS@DOMAIN.EXT" }; //ADDRESS@DOMAIN.EXT; e.g.,
 name@example.com


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

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

```
node ses_verifyemailidentity.js
```

This example code can be found here on GitHub.

Verifying a Domain identity

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

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

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

This example code can be found here on GitHub.

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

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

> **Note**
> This example imports and uses the required AWS Service V3 package clients, V3 commands, and uses the `send` method in an async/await pattern. You can create this example using V2 commands instead by making some minor changes. For details, see Using V3 commands (p. 3).

> **Note**
> Replace *AMI_ID* with the ID of the Amazon Machine Image (AMI) to run, and *KEY_PAIR_NAME* of the key pair to assign to the AMI ID.

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

// Set the parameters
const params = { Domain: "DOMAIN_NAME" }; //DOMAIN_NAME

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

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

```
node ses_verifydomainidentity.js
```

This example code can be found here on GitHub.

Deleting identities

In this example, use a Node.js module to delete email addresses or domains used with Amazon SES.

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

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

This example code can be found here on GitHub.

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

Create an object to pass the `Identity` parameter for the `DeleteIdentityCommand` method of the `SES` client class. To call the `DeleteIdentityCommand` method, create a `request` for invoking an Amazon SES client service object, passing the parameters.

> **Note**
> This example imports and uses the required AWS Service V3 package clients, V3 commands, and uses the `send` method in an async/await pattern. You can create this example using V2 commands instead by making some minor changes. For details, see Using V3 commands (p. 3).

> **Note**
> Replace *IDENTITY_TYPE* with the identity type to be deleted, and *IDENTITY_NAME* with the name of the identity to be deleted.

```
// Import required AWS SDK clients and commands for Node.js
import { DeleteIdentityCommand }  from "@aws-sdk/client-ses";
import { sesClient } from "./libs/sesClient.js";
// Set the parameters
const params = {
  IdentityType: "IDENTITY_TYPE", // IDENTITY_TYPE - i.e., 'EmailAddress' or 'Domain'
  Identity: "IDENTITY_NAME",
}; // IDENTITY_NAME

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

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 ro 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 creating a credentials file, see Loading credentials in Node.js from the shared credentials file (p. 33).

  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 (p. 56).

Listing your email templates

In this example, use a Node.js module to create an email template to use with Amazon SES.

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

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

This example code can be found here on GitHub.

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

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

Note
This example imports and uses the required AWS Service V3 package clients, V3 commands, and uses the `send` method in an async/await pattern. You can create this example using V2 commands instead by making some minor changes. For details, see Using V3 commands (p. 3).

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

// Set the parameters
const params = { MaxItems: "ITEMS_COUNT" }; //ITEMS_COUNT

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

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

This example code can be found here on GitHub.

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

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

Note

This example imports and uses the required AWS Service V3 package clients, V3 commands, and uses the `send` method in an async/await pattern. You can create this example using V2 commands instead by making some minor changes. For details, see Using V3 commands (p. 3).

Note

Replace ***`TEMPLATE_NAME`*** with the name of the template to return.

```
// Import required AWS SDK clients and commands for Node.js
import {  GetTemplateCommand }  from "@aws-sdk/client-ses";
import { sesClient } from "./libs/sesClient.js";
// Set the parameters
const params = { TemplateName: "TEMPLATE_NAME" };

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

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

```
node ses_gettemplate.js
```

This example code can be found here on GitHub.

Creating an email template

In this example, use a Node.js module to create an email template to use with Amazon SES.

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

```
import  { SESClient }  from  "@aws-sdk/client-ses";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create SES service object.
```

```
const sesClient = new SESClient({ region: REGION });
export  { sesClient };
```

This example code can be found here on GitHub.

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

Create an object to pass the parameters for the `CreateTemplateCommand` method of the `SES` client class, including `TemplateName`, `HtmlPart`, `SubjectPart`, and `TextPart`. To call the `CreateTemplateCommand` method, invoke an Amazon SES client service object, passing the parameters.

Note

This example imports and uses the required AWS Service V3 package clients, V3 commands, and uses the `send` method in an async/await pattern. You can create this example using V2 commands instead by making some minor changes. For details, see Using V3 commands (p. 3).

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 (p. 3).

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.

```
// Import required AWS SDK clients and commands for Node.js
import { CreateTemplateCommand }  from "@aws-sdk/client-ses";
import { sesClient } from "./libs/sesClient.js";
// Create createTemplate params
const params = {
  Template: {
    TemplateName: "TEMPLATE_NAME", //TEMPLATE_NAME
    HtmlPart: "HTML_CONTENT",
    SubjectPart: "SUBJECT",
    TextPart: "TEXT_CONTENT",
  },
};

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

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

```
node ses_createtemplate.js
```

This example code can be found here on GitHub.

Updating an email template

In this example, use a Node.js module to create an email template to use with Amazon SES.

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

```
import  { SESClient }  from  "@aws-sdk/client-ses";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "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 (p. 3).

> **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.

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

// Set the parameters
const params = {
  Template: {
    TemplateName: "TEMPLATE_NAME", //TEMPLATE_NAME
    HtmlPart: "HTML_CONTENT", //HTML_CONTENT; i.e., HTML content in the email
    SubjectPart: "SUBJECT_LINE", //SUBJECT_LINE; i.e., email subject line
    TextPart: "TEXT_CONTENT", //TEXT_CONTENT; i.e., body of email
  },
};

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

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

```
node ses_updatetemplate.js
```

This example code can be found here on GitHub.

Deleting an email template

In this example, use a Node.js module to create an email template to use with Amazon SES.

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

```
import  { SESClient }  from  "@aws-sdk/client-ses";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "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 (p. 3).

> **Note**
> Replace *TEMPLATE_NAME* with the name of the template to be deleted.

```
// Import required AWS SDK clients and commands for Node.js
import { DeleteTemplateCommand }  from "@aws-sdk/client-ses";
import { sesClient } from "./libs/sesClient.js";
// Set the parameters
const params = { TemplateName: "TEMPLATE_NAME" };

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

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

```
node ses_deletetemplate.js
```

This example code can be found here on GitHub.

Sending email using Amazon SES

This Node.js code example shows:

- Send a text or HTML email.
- Send emails based on an email template.
- Send bulk emails based on an email template.

The Amazon SES API provides two different ways for you to send an email, depending on how much control you want over the composition of the email message: formatted and raw. For details, see Sending formatted email using the Amazon SES API and Sending raw email using the Amazon SES API.

The scenario

In this example, you use a series of Node.js modules to send email in a variety of ways. The Node.js modules use the SDK for JavaScript to create and use email templates using these methods of the `SES` client class:

- `SendEmailCommand`
- `SendTemplatedEmailCommand`
- `SendBulkTemplatedEmailCommand`

Prerequisite tasks

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

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

- Create a shared configurations file with your user credentials. For more information about providing a credentials JSON file, see Loading credentials in Node.js from the shared credentials file (p. 33).

> **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 (p. 56).

Email message sending requirements

Amazon SES composes an email message and immediately queues it for sending. To send email using the `SendEmailCommand` method, your message must meet the following requirements:

- You must send the message from a verified email address or domain. If you attempt to send email using a non-verified address or domain, the operation results in an `"Email address not verified"` error.
- If your account is still in the Amazon SES sandbox, you can only send to verified addresses or domains, or to email addresses associated with the Amazon SES Mailbox Simulator. For more information, see Verifying email addresses and domains in the Amazon Simple Email Service Developer Guide.
- The total size of the message, including attachments, must be smaller than 10 MB.
- The message must include at least one recipient email address. The recipient address can be a To: address, a CC: address, or a BCC: address. If a recipient email address is not valid (that is, it is not in the format `UserName@[SubDomain.]Domain.TopLevelDomain`), the entire message is rejected, even if the message contains other recipients that are valid.
- The message cannot include more than 50 recipients across the To:, CC: and BCC: fields. If you need to send an email message to a larger audience, you can divide your recipient list into groups of 50 or fewer, and then call the `sendEmail` method several times to send the message to each group.

Sending an email

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

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

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

This example code can be found here on GitHub.

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

Create an object to pass the parameter values that define the email to be sent, including sender and receiver addresses, subject, and email body in plain text and HTML formats, to the `SendEmailCommand` method of the `SES` client class. To call the `SendEmailCommand` method, invoke an Amazon SES service object, passing the parameters.

> **Note**
> This example imports and uses the required AWS Service V3 package clients, V3 commands, and uses the `send` method in an async/await pattern. You can create this example using V2 commands instead by making some minor changes. For details, see Using V3 commands (p. 3).

> **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.

```
// Create the promise and SES service object

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

// Set the parameters
const params = {
```

```
  Destination: {
    /* required */
    CcAddresses: [
      /* more items */
    ],
    ToAddresses: [
      "RECEIVER_ADDRESS", //RECEIVER_ADDRESS
      /* more To-email addresses */
    ],
  },
  Message: {
    /* required */
    Body: {
      /* required */
      Html: {
        Charset: "UTF-8",
        Data: "HTML_FORMAT_BODY",
      },
      Text: {
        Charset: "UTF-8",
        Data: "TEXT_FORMAT_BODY",
      },
    },
    Subject: {
      Charset: "UTF-8",
      Data: "EMAIL_SUBJECT",
    },
  },
  Source: "SENDER_ADDRESS", // SENDER_ADDRESS
  ReplyToAddresses: [
    /* more items */
  ],
};


const run = async () => {
  try {
    const data = await sesClient.send(new SendEmailCommand(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. The email is queued for sending by Amazon SES.

```
node ses_sendemail.js
```

This example code can be found found here on GitHub.

Sending an email using a template

In this example, use a Node.js module to send email with Amazon SES. Create a Node.js module with the file name `ses_sendtemplatedemail.js`. Configure the SDK as previously shown, including installing the required clients and packages.

Create an object to pass the parameter values that define the email to be sent, including sender and receiver addresses, subject, email body in plain text and HTML formats, to the `SendTemplatedEmailCommand` method of the `SES` client class. To call the

`SendTemplatedEmailCommand` method, invoke an Amazon SES client service object, passing the parameters.

Note

This example imports and uses the required AWS Service V3 package clients, V3 commands, and uses the `send` method in an async/await pattern. You can create this example using V2 commands instead by making some minor changes. For details, see Using V3 commands (p. 3).

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

// Set the parameters
const params = {
  Destination: {
    /* required */
    CcAddresses: [
      /* more CC email addresses */
    ],
    ToAddresses: [
      "RECEIVER_ADDRESS", // RECEIVER_ADDRESS
      /* more To-email addresses */
    ],
  },
  Source: "SENDER_ADDRESS", //SENDER_ADDRESS
  Template: "TEMPLATE_NAME", // TEMPLATE_NAME
  TemplateData: '{ "REPLACEMENT_TAG_NAME":"REPLACEMENT_VALUE" }' /* required */,
  ReplyToAddresses: [],
};

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

To run the example, enter the following at the command prompt. The email is queued for sending by Amazon SES.

```
node ses_sendtemplatedemail.js
```

This example code can be found here on GitHub.

Sending bulk email using a template

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

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

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

This example code can be found here on GitHub.

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

Create an object to pass the parameter values that define the email to be sent, including sender and receiver addresses, subject, and email body in plain text and HTML formats, to the `SendBulkTemplatedEmailCommand` method of the `SES` client class. To call the `SendBulkTemplatedEmailCommand` method, invoke an Amazon SES service object, passing the parameters.

Note
This example imports and uses the required AWS Service V3 package clients, V3 commands, and uses the `send` method in an async/await pattern. You can create this example using V2 commands instead by making some minor changes. For details, see Using V3 commands (p. 3).

Note
Replace *`RECEIVER_ADDRESSES`* with the address to send the email to, and *`SENDER_ADDRESS`* with the email address to the send the email from.

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

// Set the parameters
var params = {
  Destinations: [
    /* required */
    {
      Destination: {
        /* required */
        CcAddresses: [
          "RECEIVER_ADDRESSES", //RECEIVER_ADDRESS
          /* more items */
        ],
        ToAddresses: [
          /* more items */
        ],
      },
      ReplacementTemplateData: '{ "REPLACEMENT_TAG_NAME":"REPLACEMENT_VALUE" }',
    },
  ],
  Source: "SENDER_ADDRESS", // SENDER_ADDRESS
  Template: "TEMPLATE", //TEMPLATE
  DefaultTemplateData: '{ "REPLACEMENT_TAG_NAME":"REPLACEMENT_VALUE" }',
  ReplyToAddresses: [],
};

const run = async () => {
  try {
    const data = await sesClient.send(new SendBulkTemplatedEmailCommand(params));
    console.log("Success.", data);
    return data; // For unit tests.
```

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

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.

Using IP address filters for email receipt in Amazon SES

This Node.js code example shows:

- How to create IP address filters to accept or reject mail that originates from an IP address or range of IP addresses.
- How to list your current IP address filters.
- How to delete an IP address filter.

In Amazon SES, a *filter* is a data structure that consists of a name, an IP address range, and the functionality to allow or block mail from it. IP addresses you want to block or allow are specified as a single IP address or a range of IP addresses in Classless Inter-Domain Routing (CIDR) notation. For details on how Amazon SES receives email, see Amazon SES email-receiving concepts in the Amazon Simple Email Service Developer Guide.

The scenario

In this example, a series of Node.js modules are used 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:

- `CreateReceiptFilterCommand`
- `ListReceiptFiltersCommand`
- `DeleteReceiptFilterCommand`

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 Loading credentials in Node.js from the shared credentials file (p. 33).

> **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 (p. 56).

Creating an IP address filter

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

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

```
import  { SESClient }  from  "@aws-sdk/client-ses";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "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_createreceiptfilter.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 IP filter, including the filter name, an IP address or range of addresses to filter, and whether to allow or block email traffic from the filtered addresses. To call the `CreateReceiptFilterCommand` 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 (p. 3).

> **Note**
> Replace *IP_ADDRESS_OR_RANGE* with the IP address or range of addresses to filter, *POLICY* with with `ALLOW` or `BLOCK`, and *NAME* with the filter name.

```
// Import required AWS SDK clients and commands for Node.js
import { CreateReceiptFilterCommand } from "@aws-sdk/client-ses";
import { sesClient } from "./libs/sesClient.js";
// Set the parameters
const params = {
  Filter: {
    IpFilter: {
      Cidr: "IP_ADDRESS_OR_RANGE", // (in code; either a single IP address (10.0.0.1) or an
 IP address range in CIDR notation (10.0.0.1/24)),
      Policy: "POLICY", // 'ALLOW' or 'BLOCK' email traffic from the filtered
 addressesOptions.
    },
    Name: "NAME" // NAME (the filter name)
```

```
  },
};

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

To run the example, enter the following at the command prompt. The filter is created in Amazon SES.

```
node ses_createreceiptfilter.js
```

This example code can be found here on GitHub.

Listing your IP address filters

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

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

```
import  { SESClient }  from  "@aws-sdk/client-ses";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "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_listreceiptfilters.js`. Configure the SDK as previously shown, including installing the required clients and packages.

Create an empty parameters object. To call the `ListReceiptFiltersCommand` method, invoking an Amazon SES service object, passing the parameters.

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

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

To run the example, enter the following at the command prompt. Amazon SES returns the filter list.

```
node ses_listreceiptfilters.js
```

This example code can be found here on GitHub.

Deleting an IP address filter

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

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

```
import  { SESClient }  from  "@aws-sdk/client-ses";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "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_deletereceiptfilter.js`. Configure the SDK as previously shown, including installing the required clients and packages.

Create an object to pass the name of the IP filter to delete. To call the `DeleteReceiptFilterCommand` 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 (p. 3).

> **Note**
> Replace ***FILTER_NAME*** with the name of the IP filter to delete.

```
// Import required AWS SDK clients and commands for Node.js
import {
  DeleteReceiptFilterCommand
}  from "@aws-sdk/client-ses";
import { sesClient } from "./libs/sesClient.js";
// Set the parameters
const params = { FilterName: "FILTER_NAME" }; //FILTER_NAME

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

To run the example, enter the following at the command prompt. The filter is deleted from Amazon SES.

```
node ses_deletereceiptfilter.js
```

This example code can be found here on GitHub.

Using receipt rules in Amazon SES

This Node.js code example shows:

- How to create and delete receipt rules.
- How to organize receipt rules into receipt rule sets.

Receipt rules in Amazon SES specify what to do with email received for email addresses or domains you own. A *receipt rule* contains a condition and an ordered list of actions. If the recipient of an incoming email matches a recipient specified in the conditions for the receipt rule, Amazon SES performs the actions that the receipt rule specifies.

To use Amazon SES as your email receiver, you must have at least one active *receipt rule set*. A receipt rule set is an ordered collection of receipt rules that specify what Amazon SES should do with mail it receives across your verified domains. For more information, see Creating receipt rules for Amazon SES email receiving and Creating a receipt rule set for Amazon SES email receiving in the Amazon Simple Email Service Developer Guide.

The scenario

In this example, a series of Node.js modules are used 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:

- `CreateReceiptRuleCommand`
- `DeleteReceiptRuleCommand`
- `CreateReceiptRuleSetCommand`
- `DeleteReceiptRuleSetCommand`

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 credentials JSON file, see Loading credentials in Node.js from the shared credentials file (p. 33).

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 (p. 56).

Creating an Amazon S3 receipt rule

Each receipt rule for Amazon SES contains an ordered list of actions. This example creates a receipt rule with an Amazon S3 action, which delivers the mail message to an Amazon S3 bucket. For details on receipt rule actions, see Action options in the Amazon Simple Email Service Developer Guide.

For Amazon SES to write email to an Amazon S3 bucket, create a bucket policy that gives `PutObject` permission to Amazon SES. For information about creating this bucket policy, see Give Amazon SES permission to write to your Amazon S3 bucket in the Amazon Simple Email Service Developer Guide.

In this example, use a Node.js module to create a receipt rule in Amazon SES to save received messages in an Amazon S3 bucket.

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 = "REGION"; //e.g. "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_createreceiptrule.js`. Configure the SDK as previously shown.

Create a parameters object to pass the values needed to create for the receipt rule set. To call the `CreateReceiptRuleSetCommand` method, invoke an Amazon SES service object, passing the parameters.

> **Note**
> Replace *`S3_BUCKET_NAME`* with the name of the Amazon S3 bucket, *`EMAIL_ADDRESS | DOMAIN`* with the email, or domain, *`RULE_NAME`* with the name of the rule, and *`RULE_SET_NAME`* with the name of the ruleset.

```
// Import required AWS SDK clients and commands for Node.js
import { CreateReceiptRuleCommand }  from "@aws-sdk/client-ses";
import { sesClient } from "./libs/sesClient.js";
// Set the parameters
const params = {
  Rule: {
    Actions: [
      {
        S3Action: {
          BucketName: "BUCKET_NAME", // S3_BUCKET_NAME
          ObjectKeyPrefix: "email",
        },
      },
    ],
    Recipients: [
      "EMAIL_ADDRESS", // The email addresses, or domain
      /* more items */
    ],
    Enabled: true | false,
    Name: "RULE_NAME", // RULE_NAME
    ScanEnabled: true | false,
    TlsPolicy: "Optional",
```

```
  },
  RuleSetName: "RULE_SET_NAME", // RULE_SET_NAME
};

const run = async () => {
  try {
    const data = await sesClient.send(new CreateReceiptRuleCommand(params));
    console.log("Rule created", data);
    return data; // For unit tests.
  } catch (err) {
    console.log("Error", err.stack);
  }
};
run();
```

To run the example, enter the following at the command prompt. Amazon SES creates the receipt rule.

```
node ses_createreceiptrule.js
```

This example code can be found here on GitHub.

Deleting a receipt rule

In this example, use a Node.js module to send email with Amazon SES. Create a Node.js module with the file name `ses_deletereceiptrule.js`. Configure the SDK as previously shown, including installing the required clients and packages.

Create a parameters object to pass the name for the receipt rule to delete. To call the `DeleteReceiptRuleCommand` method, invoke an Amazon SES service object, passing the parameters.

> **Note**
> Replace *`RULE_NAME`* with the name of the rule, and *`RULE_SET_NAME`* with the rule set name.

```
// Import required AWS SDK clients and commands for Node.js
import { DeleteReceiptRuleCommand }  from "@aws-sdk/client-ses";
import { sesClient } from "./libs/sesClient.js";
// Set the deleteReceiptRule params
var params = {
  RuleName: "RULE_NAME", // RULE_NAME
  RuleSetName: "RULE_SET_NAME", // RULE_SET_NAME
};

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

To run the example, enter the following at the command prompt. Amazon SES creates the receipt rule set list.

```
node ses_deletereceiptrule.js
```

This example code can be found here on GitHub.

Creating a receipt rule set

In this example, use a Node.js module to send email with Amazon SES. Create a Node.js module with the file name `ses_createreceiptruleset.js`. Configure the SDK as previously shown, including installing the required clients and packages.

Create a parameters object to pass the name for the new receipt rule set. To call the `CreateReceiptRuleSetCommand` method, invoke an Amazon SES client service object, passing the parameters.

Note
Replace *REGION* with your AWS Region, *RULE_SET_NAME* with the rule set name.

```
// Import required AWS SDK clients and commands for Node.js
import {
  CreateReceiptRuleSetCommand
} from "@aws-sdk/client-ses";
import { sesClient } from "./libs/sesClient.js";
// Set the parameters
const params = { RuleSetName: "RULE_SET_NAME" }; //RULE_SET_NAME

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

To run the example, enter the following at the command prompt. Amazon SES creates the receipt rule set list.

```
node ses_createreceiptruleset.js
```

This example code can be found here on GitHub.

Deleting a receipt rule set

In this example, use a Node.js module to send email with Amazon SES. Create a Node.js module with the file name `ses_deletereceiptruleset.js`. Configure the SDK as previously shown, including installing the required clients and packages.

Create an object to pass the name for the receipt rule set to delete. To call the `DeleteReceiptRuleSetCommand` method, invoke an Amazon SES client service object, passing the parameters.

Note
Replace *RULE_SET_NAME* with the rule set name.

```
// Import required AWS SDK clients and commands for Node.js
import { DeleteReceiptRuleSetCommand }  from "@aws-sdk/client-ses";
import { sesClient } from "./libs/sesClient.js";
// Set the parameters
const params = { RuleSetName: "RULE_SET_NAME" }; //RULE_SET_NAME
```

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

To run the example, enter the following at the command prompt. Amazon SES creates the receipt rule set list.

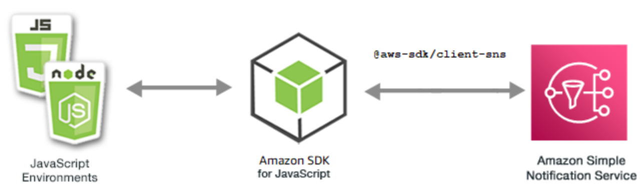
```
node ses_deletereceiptruleset.js // If you prefer JavaScript, enter 'node
 ses_deletereceiptruleset.js'
```

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

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 credentials JSON file, see Loading credentials in Node.js from the shared credentials file (p. 33).

> **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 (p. 56).

Creating a Topic

In this example, use a Node.js module to create an Amazon SNS topic.

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

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

This example code can be found here on GitHub.

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

Create an object to pass the `Name` for the new topic to the `CreateTopicCommand` method of the `SNS` client class. To call the `CreateTopicCommand` method, create an asynchronous function invoking an Amazon SNS service object, passing the parameters object. The `data` returned contains the ARN of the topic.

Note
Replace *`TOPIC_NAME`* with the name of the topic.

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

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

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

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

```
node sns_createtopic.js
```

This example code can be found here on GitHub.

Listing Your Topics

In this example, use a Node.js module to list all Amazon SNS topics.

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

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

This example code can be found here on GitHub.

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

Create an empty object to pass to the `ListTopicsCommand` method of the `SNS` client class. To call the `ListTopicsCommand` method, create an asynchronous function invoking an Amazon SNS service object, passing the parameters object. The `data` returned contains an array of your topic Amazon Resource Names (ARNs).

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

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

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

```
node sns_listtopics.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";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create SNS service object.
const snsClient = new SNSClient({ region: REGION });
export  { snsClient };
```

This example code can be found here on GitHub.

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

Create an object containing the `TopicArn` of the topic to delete to pass to the `DeleteTopicCommand` method of the `SNS` client class. To call the `DeleteTopicCommand` method, create an asynchronous function invoking an Amazon SNS client service object, passing the parameters object.

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

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

// Import required AWS SDK clients and commands for Node.js
import {DeleteTopicCommand } from "@aws-sdk/client-sns";
import {snsClient } from "./libs/snsClient.js";

// Set the parameters
const params = { TopicArn: "TOPIC_ARN" }; //TOPIC_ARN

const run = async () => {
  try {
    const data = await snsClient.send(new DeleteTopicCommand(params));
    console.log("Success.",  data);
```

```
    return data; // For unit tests.
  } catch (err) {
    console.log("Error", err.stack);
  }
};
run();
```

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

```
node sns_deletetopic.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";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create SNS service object.
const snsClient = new SNSClient({ region: REGION });
export  { snsClient };
```

This example code can be found here on GitHub.

Create a Node.js module with the file name `sns_gettopicattributes.js`. Configure the SDK as previously shown.

Create an object containing the `TopicArn` of a topic to delete to pass to the `GetTopicAttributesCommand` method of the `SNS` client class. To call the `GetTopicAttributesCommand` method, invoking an Amazon SNS client service object, passing the parameters object.

> **Note**
> Replace *TOPIC_ARN* with the ARN of the topic.

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

// Set the parameters
const params = { TopicArn: "TOPIC_ARN" }; // TOPIC_ARN

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

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

```
node sns_gettopicattributes.js
```

This example code can be found here on GitHub.

Setting Topic Attributes

In this example, use a Node.js module to set the mutable attributes of an Amazon SNS topic.

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

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

This example code can be found here on GitHub.

Create a Node.js module with the file name `sns_settopicattributes.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 cleint service object, passing the parameters object.

> **Note**
> Replace *ATTRIBUTE_NAME* with the name of the attribute you are setting, *TOPIC_ARN* with the Amazon Resource Name (ARN) of the topic whose attributes you want to set, and *NEW_ATTRIBUTE_VALUE* with the new value for that attribute.

```

// Import required AWS SDK clients and commands for Node.js
import {SetTopicAttributesCommand } from "@aws-sdk/client-sns";
import {snsClient } from "./libs/snsClient.js";

// Set the parameters
const params = {
  AttributeName: "ATTRIBUTE_NAME", // ATTRIBUTE_NAME
  TopicArn: "TOPIC_ARN", // TOPIC_ARN
  AttributeValue: "NEW_ATTRIBUTE_VALUE", //NEW_ATTRIBUTE_VALUE
};

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

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

```
node sns_settopicattributes.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 credentials JSON file, see Loading credentials in Node.js from the shared credentials file (p. 33).

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 (p. 56).

Publishing a Message to an SNS Topic

In this example, use a Node.js module to publish a message to an Amazon SNS topic.

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

```
import  { SNSClient } from "@aws-sdk/client-sns";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
```

```
// Create SNS service object.
const snsClient = new SNSClient({ region: REGION });
export  { snsClient };
```

This example code can be found here on GitHub.

Create a Node.js module with the file name `sns_publishtotopic.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 SNStopic. For details on available SMS attributes, see SetSMSAttributes.

Pass the parameters to the `PublishCommand` method of the `SNS` client class. create an asynchronous function invoking an Amazon SNS client service object, passing the parameters object.

> **Note**
> Replace *`MESSAGE_TEXT`* with the message text, and *`TOPIC_ARN`* with the ARN of the SNS topic.

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

// Set the parameters
var params = {
  Message: "MESSAGE_TEXT", // MESSAGE_TEXT
  TopicArn: "TOPIC_ARN", //TOPIC_ARN
};

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

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

```
node sns_publishtotopic.js
```

This example code can be found here on GitHub.

Managing Subscriptions in Amazon SNS

This Node.js code example shows:

- How to list all subscriptions to an Amazon SNS topic.
- How to subscribe an email address, an application endpoint, or an AWS Lambda function to an Amazon SNS topic.

- How to unsubscribe from Amazon SNS topics.

The Scenario

In this example, you use a series of Node.js modules to publish notification messages to Amazon SNS topics. The Node.js modules use the SDK for JavaScript to manage topics using these methods of the `SNS` client class:

- `ListSubscriptionsByTopicCommand`
- `SubscribeCommand`
- `ConfirmSubscriptionCommand`
- `UnsubscribeCommand`

Prerequisite Tasks

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

- Set up the project environment to run these Node TypeScript examples, and install the required AWS SDK for JavaScript and third-party modules. Follow the instructions on GitHub.
- Create a shared configurations file with your user credentials. For more information about providing a credentials JSON file, see Loading credentials in Node.js from the shared credentials file (p. 33).

> **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 (p. 56).

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";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create SNS service object.
const snsClient = new SNSClient({ region: REGION });
export  { snsClient };
```

This example code can be found here on GitHub.

Create a Node.js module with the file name `sns_listsubscriptions.js`. Configure the SDK as previously shown.

Create an object containing the `TopicArn` parameter for the topic whose subscriptions you want to list. Pass the parameters to the `ListSubscriptionsByTopicCommand` method of the `SNS` client class. To call the `ListSubscriptionsByTopicCommand` method, create an asynchronous function invoking an Amazon SNS client service object, and passing the parameters object.

Note

Replace *TOPIC_ARN* with the Amazon Resource Name (ARN) for the topic whose subscriptions you want to list .

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

// Set the parameters
const params = { TopicArn: "TOPIC_ARN" }; //TOPIC_ARN

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

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

```
node sns_listsubscriptions.js
```

This example code can be found here on GitHub.

Subscribing an Email Address to a Topic

In this example, use a Node.js module to subscribe an email address so that it receives SMTP email messages from an Amazon SNS topic.

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

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

This example code can be found here on GitHub.

Create a Node.js module with the file name `sns_subscribeemail.js`. Configure the SDK as previously shown.

Create an object containing the `Protocol` parameter to specify the `email` protocol, the `TopicArn` for the topic to subscribe to, and an email address as the message `Endpoint`. Pass the parameters to the `SubscribeCommand` method of the `SNS` client class. You can use the `subscribe` method to subscribe several different endpoints to an Amazon SNS topic, depending on the values used for parameters passed, as other examples in this topic will show.

To call the `SubscribeCommand` method, create an asynchronous function invoking an Amazon SNS client service object, and passing the parameters object.

Note
Replace *TOPIC_ARN* with the Amazon Resource Name (ARN) for the topic, and *EMAIL_ADDRESS* with the email address to subcribe to.

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

// Set the parameters
const params = {
  Protocol: "email" /* required */,
  TopicArn: "TOPIC_ARN", //TOPIC_ARN
  Endpoint: "EMAIL_ADDRESS", //EMAIL_ADDRESS
};

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

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

```
node sns_subscribeemail.js
```

This example code can be found here on GitHub.

Confirming Subscriptions

In this example, use a Node.js module to verify an endpoint owner's intent to receive emails by validating the token sent to the endpoint by a previous subscribe action.

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

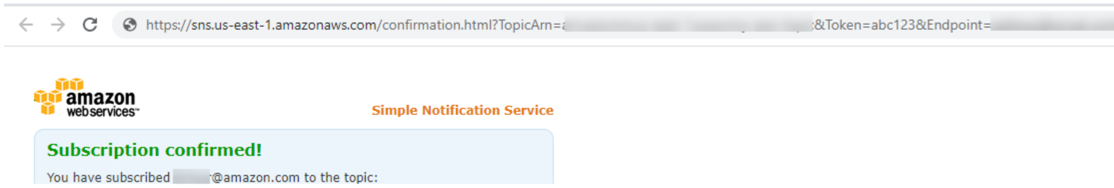
```
import  { SNSClient } from "@aws-sdk/client-sns";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create SNS service object.
const snsClient = new SNSClient({ region: REGION });
export  { snsClient };
```

This example code can be found here on GitHub.

Create a Node.js module with the file name `sns_confirmsubscription.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 `AutheticateOnUnsubscribe`. If set to `TRUE` the `Confirm Subscription` action requires an AWS signature.

The token is a short-lived token sent to the owner of an endpoint during a previous `SUBSCRIBE` action. For example, for an email endpoint the `TOKEN` is in the URL of the Confirm Subscription email sent to the email owner. For example, `abc123` is the token in the following URL.



To call the `ConfirmSubscriptionCommand` method, create an asynchronous function invoking an Amazon SNS client service object, passing the parameters object.

> **Note**
> Replace *`TOPIC_ARN`* with the Amazon Resource Name (ARN) for the topic, *`TOKEN`* with the token value from the URL sent to the endpoint owner in a previous `Subscribe` action, and define *`AuthenticateOnUnsubscribe`*. with a value of `TRUE` or `FALSE`.

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

// Set the parameters
const params = {
  Token: "TOKEN", // Required. Token sent to the endpoint by an earlier Subscribe action.
 */
  TopicArn: "TOPIC_ARN", // Required
  AuthenticateOnUnsubscribe: "true", // 'true' or 'false'
};

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

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

```
node sns_confirmsubscription.js
```

This example code can be found here on GitHub.

Subscribing an Application Endpoint to a Topic

In this example, use a Node.js module to subscribe a mobile application endpoint so it receives notifications from an Amazon SNS topic.

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

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

This example code can be found here on GitHub.

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

Create an object containing the `Protocol` parameter to specify the `application` protocol, the `TopicArn` for the topic to subscribe to, and the Amazon Resource Name (ARN) of a mobile application endpoint for the `Endpoint` parameter. Pass the parameters to the `SubscribeCommand` method of the `SNS` client class.

To call the `SubscribeCommand` method, create an asynchronous function invoking an Amazon SNS service object, passing the parameters object.

Note

Replace *`TOPIC_ARN`* with the Amazon Resource Name (ARN) for the topic, and *`MOBILE_ENDPOINT_ARN`* with the endpoint you are subscribing to the topic.

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

// Set the parameters
const params = {
  Protocol: "application" /* required */,
  TopicArn: "TOPIC_ARN", //TOPIC_ARN
  Endpoint: "MOBILE_ENDPOINT_ARN", // MOBILE_ENDPOINT_ARN
};

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

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

```
node sns_subscribeapp.js
```

This example code can be found here on GitHub.

Subscribing a Lambda Function to a Topic

In this example, use a Node.js module to subscribe an AWS Lambda function so it receives notifications from an Amazon SNS topic.

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

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

This example code can be found here on GitHub.

Create a Node.js module with the file name `sns_subscribelambda.js`. Configure the SDK as previously shown.

Create an object containing the `Protocol` parameter, specifying the `lambda` protocol, the `TopicArn` for the topic to subscribe to, and the Amazon Resource Name (ARN) of an AWS Lambda function as the `Endpoint` parameter. Pass the parameters to the `SubscribeCommand` method of the `SNS` client class.

To call the `SubscribeCommand` method, create an asynchronous function invoking an Amazon SNS client service object, passing the parameters object.

> **Note**
> Replace *`TOPIC_ARN`* with the Amazon Resource Name (ARN) for the topic, and *`LAMBDA_FUNCTION_ARN`* with the Amazon Resource Name (ARN) of the Lambda function.

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

// Set the parameters
const params = {
  Protocol: "lambda" /* required */,
  TopicArn: "TOPIC_ARN", //TOPIC_ARN
  Endpoint: "LAMBDA_FUNCTION_ARN", //LAMBDA_FUNCTION_ARN
};

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

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

```
node sns_subscribelambda.js
```

This example code can be found here on GitHub.

Unsubscribing from a Topic

In this example, use a Node.js module to unsubscribe an Amazon SNS topic subscription.

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

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

This example code can be found here on GitHub.

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

Create an object containing the `SubscriptionArn` parameter, specifying the Amazon Resource Name (ARN) of the subscription to unsubscribe. Pass the parameters to the `UnsubscribeCommand` method of the `SNS` client class.

To call the `UnsubscribeCommand` method, create an asynchronous function invoking an Amazon SNS client service object, passing the parameters object.

> **Note**
> Replace *`TOPIC_SUBSCRIPTION_ARN`* with the Amazon Resource Name (ARN) of the subscription to unsubscribe.

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

// Set the parameters
const params = { SubscriptionArn: "TOPIC_SUBSCRIPTION_ARN" }; //TOPIC_SUBSCRIPTION_ARN

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

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

```
node sns_unsubscribe.js
```

This example code can be found here on GitHub.

Sending SMS Messages with Amazon SNS

This Node.js code example shows:

- How to get and set SMS messaging preferences for Amazon SNS.
- How to check a phone number to see if it has opted out of receiving SMS messages.
- How to get a list of phone numbers that have opted out of receiving SMS messages.
- How to send an SMS message.

The Scenario

You can use Amazon SNS to send text messages, or SMS messages, to SMS-enabled devices. You can send a message directly to a phone number, or you can send a message to multiple phone numbers at once by subscribing those phone numbers to a topic and sending your message to the topic.

In this example, you use a series of Node.js modules to publish SMS text messages from Amazon SNS to SMS-enabled devices. The Node.js modules use the SDK for JavaScript to publish SMS messages using these methods of the `SNS` client class:

- `GetSMSAttributesCommand`
- `SetSMSAttributesCommand`
- `CheckIfPhoneNumberIsOptedOutCommand`
- `ListPhoneNumbersOptedOutCommand`
- `PublishCommand`

Prerequisite Tasks

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

- Set up the project environment to run these Node TypeScript examples, and install the required AWS SDK for JavaScript and third-party modules. Follow the instructions on GitHub.
- Create a shared configurations file with your user credentials. For more information about providing a credentials JSON file, see Loading credentials in Node.js from the shared credentials file (p. 33).

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 (p. 56).

Getting SMS Attributes

Use Amazon SNS to specify preferences for SMS messaging, such as how your deliveries are optimized (for cost or for reliable delivery), your monthly spending limit, how message deliveries are logged, and whether to subscribe to daily SMS usage reports. These preferences are retrieved and set as SMS attributes for Amazon SNS.

In this example, use a Node.js module to get the current SMS attributes in Amazon SNS.

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

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

This example code can be found here on GitHub.

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

Configure the SDK as previously shown, including downloading the required clients and packages. Create an object containing the parameters for getting SMS attributes, including the names of the individual attributes to get. For details on available SMS attributes, see SetSMSAttributes in the Amazon Simple Notification Service API Reference.

This example gets the `DefaultSMSType` attribute, which controls whether SMS messages are sent as `Promotional`, which optimizes message delivery to incur the lowest cost, or as `Transactional`, which optimizes message delivery to achieve the highest reliability. Pass the parameters to the `SetTopicAttributesCommand` method of the `SNS` client class. To call the `SetSMSAttributesCommand` method, create an asynchronous function invoking an Amazon SNS client service object, passing the parameters object.

> **Note**
> Replace *ATTRIBUTE_NAME* with the name of the attribute.

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

// Set the parameters
var params = {
  attributes: [
    "DefaultSMSType",
    "ATTRIBUTE_NAME",
    /* more items */
  ],
};

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

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

```
node sns_getsmstype.js
```

This example code can be found here on GitHub.

Setting SMS Attributes

In this example, use a Node.js module to get the current SMS attributes in Amazon SNS.

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

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

This example code can be found here on GitHub.

Create a Node.js module with the file name `sns_setsmstype.js`. Configure the SDK as previously shown, including installing the required clients and packages. Create an object containing the parameters for setting SMS attributes, including the names of the individual attributes to set and the values to set for each. For details on available SMS attributes, see SetSMSAttributes in the Amazon Simple Notification Service API Reference.

This example sets the `DefaultSMSType` attribute to `Transactional`, which optimizes message delivery to achieve the highest reliability. Pass the parameters to the `SetTopicAttributesCommand` method of the `SNS` client class. To call the `SetSMSAttributesCommand` method, create an asynchronous function invoking an Amazon SNS client service object, passing the parameters object.

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

// Set the parameters
const params = {
  attributes: {
    /* required */
    DefaultSMSType: "Transactional" /* highest reliability */,
    //'DefaultSMSType': 'Promotional' /* lowest cost */
  },
};

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

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

```
node sns_setsmstype.js
```

This example code can be found here on GitHub.

Checking If a Phone Number Has Opted Out

In this example, use a Node.js module to check a phone number to see if it has opted out from receiving SMS messages.

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

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

This example code can be found here on GitHub.

Create a Node.js module with the file name `sns_checkphoneoptout.js`. Configure the SDK as previously shown. Create an object containing the phone number to check as a parameter.

This example sets the `PhoneNumber` parameter to specify the phone number to check. Pass the object to the `CheckIfPhoneNumberIsOptedOutCommand` method of the `SNS` client class. To call the

`CheckIfPhoneNumberIsOptedOutCommand` method, create an asynchronous function invoking an Amazon SNS client service object, passing the parameters object.

> **Note**
>
> 1.
>
> Replace *PHONE_NUMBER* with the phone number.

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

// Set the parameters
const params = { phoneNumber: "353861230764" }; //PHONE_NUMBER, in the E.164 phone number
 structure

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

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

```
node sns_checkphoneoptout.js
```

This example code can be found here on GitHub.

Listing Opted-Out Phone Numbers

In this example, use a Node.js module to get a list of phone numbers that have opted out from receiving SMS messages.

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

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

This example code can be found here on GitHub.

Create a Node.js module with the file name `sns_listnumbersoptedout.js`. Configure the SDK as previously shown. Create an empty object as a parameter.

Pass the object to the `ListPhoneNumbersOptedOutCommand` method of the `SNS` client class. To call the `ListPhoneNumbersOptedOutCommand` method, create an asynchronous function invoking an Amazon SNS client service object, passing the parameters object.

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

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

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

```
node sns_listnumbersoptedout.js
```

This example code can be found here on GitHub.

Publishing an SMS Message

In this example, use a Node.js module to send an SMS message to a phone number.

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

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

This example code can be found here on GitHub.

Create a Node.js module with the file name `sns_publishsms.js`. Configure the SDK as previously shown, including installing the required clients and packages. Create an object containing the `Message` and `PhoneNumber` parameters.

When you send an SMS message, specify the phone number using the E.164 format. E.164 is a standard for the phone number structure used for international telecommunication. Phone numbers that follow this format can have a maximum of 15 digits, and they are prefixed with the plus character (+) and the country code. For example, a US phone number in E.164 format would appear as +1001XXX5550100.

This example sets the `PhoneNumber` parameter to specify the phone number to send the message. Pass the object to the `PublishCommand` method of the `SNS` client class. To call the `PublishCommand` method, create an asynchronous function invoking an Amazon SNS service object, passing the parameters object.

Note
Replace *`TEXT_MESSAGE`* with the text message, and *`PHONE_NUMBER`* with the phone number.

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

```
// Set the parameters
const params = {
  Message: "MESSAGE_TEXT" /* required */,
  PhoneNumber: "PHONE_NUMBER", //PHONE_NUMBER, in the E.164 phone number structure
};

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

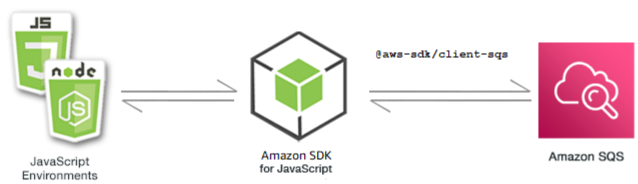
To run the example, enter the following at the command prompt.

```
node sns_publishsms.js
```

This example code can be found here on GitHub.

Amazon SQS examples

Amazon Simple Queue Service (SQS) is a fast, reliable, scalable, fully managed message queuing service. Amazon SQS lets you decouple the components of a cloud application. Amazon SQS includes standard queues with high throughput and at-least-once processing, and FIFO queues that provide first-in, first-out (FIFO) delivery and exactly-once processing.



The JavaScript API for Amazon SQS is exposed through the `SQS` client class. For more information about using the Amazon SQS; client class, see Class: SQS in the API Reference.

Topics

Using queues in Amazon SQS

This Node.js code example shows:

- How to get a list of all of your message queues.
- How to obtain the URL for a particular queue.
- How to create and delete queues.

About the example

In this example, a series of Node.js modules are used to work with queues. The Node.js modules use the SDK for JavaScript to enable queues to call the following methods of the `SQS` client class:

- `ListQueuesCommand`
- `CreateQueueCommand`
- `GetQueueUrlCommand`
- `DeleteQueueCommand`

For more information about Amazon SQS messages, see How queues work in the *Amazon Simple Queue Service 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 Loading credentials in Node.js from the shared credentials file (p. 33).

> **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 (p. 56).

Listing your queues

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

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

This example code can be found here on GitHub.

Create a Node.js module with the file name `sqs_listqueues.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. Create a JSON object containing the parameters needed to list your queues, which by default is an empty object. Call the `ListQueuesCommand` method to retrieve the list of queues. The function returns the URLs of all queues.

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

const run = async () => {
  try {
    const data = await sqsClient.send(new ListQueuesCommand({}));
    console.log("Success", data);
    return data; // For unit tests.
  } catch (err) {
    console.error(err, err.stack);
  }
};
run();
```

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

```
node sqs_listqueues.js
```

This example code can be found here on GitHub.

Creating a queue

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

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

This example code can be found here on GitHub.

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

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

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

This example code can be found here on GitHub.

Create a Node.js module with the file name `sqs_createqueue.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. Create a JSON object containing the parameters needed to list your queues, which must include the name for the queue created. The parameters can also contain attributes for the queue, such as the number of seconds for which message delivery is delayed or the number of seconds to retain a received message. Call the `CreateQueueCommand` method. The function returns the URL of the created queue.

> **Note**
> Replace *`SQS_QUEUE_NAME`* with the name of the Amazon SNS queue, *`DelaySeconds`* with the number of seconds for which message delivery is delayed, and *`MessageRetentionPeriod`* with the number of seconds to retain a received message.

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

// Set the parameters
const params = {
  QueueName: "SQS_QUEUE_NAME", //SQS_QUEUE_URL
  Attributes: {
    DelaySeconds: "60", // Number of seconds delay.
    MessageRetentionPeriod: "86400", // Number of seconds delay.
  },
};

const run = async () => {
  try {
    const data = await sqsClient.send(new CreateQueueCommand(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 sqs_createqueue.js
```

This example code can be found here on GitHub.

Getting the URL for a queue

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

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

This example code can be found here on GitHub.

Create a Node.js module with the file name `sqs_getqueueurl.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. Create a JSON object containing the parameters needed to list your queues, which must include the name of the queue whose

URL you want. Call the `GetQueueUrlCommand` method. The function returns the URL of the specified queue.

Note
Replace and *SQS_QUEUE_NAME* with the SQS queue name.

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

// Set the parameters
const params = { QueueName: "SQS_QUEUE_NAME" };

const run = async () => {
  try {
    const data = await sqsClient.send(new GetQueueUrlCommand(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 sqs_getqueueurl.js
```

This example code can be found here on GitHub.

Deleting a queue

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

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

This example code can be found here on GitHub.

Create a Node.js module with the file name `sqs_deletequeue.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. Create a JSON object containing the parameters needed to delete a queue, which consists of the URL of the queue you want to delete. Call the `DeleteQueueCommand` method.

Note
Replace *SQS_QUEUE_URL* with the URL of the Amazon SQS queue.

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

// Set the parameters
const params = { QueueUrl: "SQS_QUEUE_URL" }; //SQS_QUEUE_URL e.g., 'https://
sqs.REGION.amazonaws.com/ACCOUNT-ID/QUEUE-NAME'
```

```
const run = async () => {
  try {
    const data = await sqsClient.send(new DeleteQueueCommand(params));
    console.log("Success", data);
    return data; // For unit tests.
  } catch (err) {
    console.error(err, err.stack);
  }
};
run();
```

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

```
node sqs_deletequeue.js
```

This example code can be found here on GitHub.

Sending and receiving messages in Amazon SQS

This Node.js code example shows:

- How to send messages in a queue.
- How to receive messages in a queue.
- How to delete messages in a queue.

The scenario

In this example, a series of Node.js modules are used to send and receive messages. The Node.js modules use the SDK for JavaScript to send and receive messages by using these methods of the `SQS` client class:

- `SendMessageCommand`
- `ReceiveMessageCommand`
- `DeleteMessageCommand`

For more information about Amazon SQS messages, see Sending a message to an Amazon SQS queue and Receiving and deleting a message from an Amazon SQS queue in the *Amazon Simple Queue Service 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 Loading credentials in Node.js from the shared credentials file (p. 33).
- Create an Amazon SQS queue. For an example of creating a queue, see Using queues in Amazon SQS (p. 268).

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 (p. 56).

Sending a message to a queue

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

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

This example code can be found here on GitHub.

Create a Node.js module with the file name `sqs_sendmessage.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. Create a JSON object containing the parameters needed for your message, including the URL of the queue to which you want to send this message. In this example, the message provides details about a book on a list of fiction best sellers including the title, author, and number of weeks on the list.

Call the `SendMessageCommand` method. The callback returns the unique ID of the message.

Note
Replace *`SQS_QUEUE_URL`* with the URL of the SQS queue.

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

// Set the parameters
const params = {
  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_QUEUE_URL; e.g., 'https://sqs.REGION.amazonaws.com/
ACCOUNT-ID/QUEUE-NAME'
};

const run = async () => {
  try {
    const data = await sqsClient.send(new SendMessageCommand(params));
    console.log("Success, message sent. MessageID:", data.MessageId);
    return data; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

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

```
node sqs_sendmessage.js
```

This example code can be found here on GitHub.

Receiving and deleting messages from a queue

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

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

This example code can be found here on GitHub.

Create a Node.js module with the file name `sqs_receivemessage.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. Create a JSON object containing the parameters needed for your message, including the URL of the queue from which you want to receive messages. In this example, the parameters specify receipt of all message attributes, as well as receipt of no more than 10 messages.

Call the `ReceiveMessageCommand` method. The callback returns an array of `Message` objects from which you can retrieve `ReceiptHandle` for each message that you use to later delete that message. Create another JSON object containing the parameters needed to delete the message, which are the URL of the queue and the `ReceiptHandle` value. Call the `DeleteMessageCommand` method to delete the message you received.

> **Note**
> Replace and *SQS_QUEUE_URL* with the URL of the SQS queue.

```

// Import required AWS SDK clients and commands for Node.js
import {
  ReceiveMessageCommand,
  DeleteMessageCommand,
} from  "@aws-sdk/client-sqs";
import { sqsClient } from  "./libs/sqsClient.js";

// Set the parameters
```

```
const queueURL = "SQS_QUEUE_URL"; //SQS_QUEUE_URL; e.g., 'https://sqs.REGION.amazonaws.com/
ACCOUNT-ID/QUEUE-NAME'
const params = {
  AttributeNames: ["SentTimestamp"],
  MaxNumberOfMessages: 10,
  MessageAttributeNames: ["All"],
  QueueUrl: queueURL,
  VisibilityTimeout: 20,
  WaitTimeSeconds: 0,
};

const run = async () => {
  try {
    const data = await sqsClient.send(new ReceiveMessageCommand(params));
    if (data.Messages) {
      var deleteParams = {
        QueueUrl: queueURL,
        ReceiptHandle: data.Messages[0].ReceiptHandle,
      };
      try {
        const data = await sqsClient.send(new DeleteMessageCommand(deleteParams));
        console.log("Message deleted", data);
      } catch (err) {
        console.log("Error", err);
      }
    } else {
      console.log("No messages to delete");
    }
    return data; // For unit tests.
  } catch (err) {
    console.log("Receive Error", err);
  }
};
run();
```

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

```
node sqs_receivemessage.js
```

This example code can be found here on GitHub.

Managing visibility timeout in Amazon SQS

This Node.js code example shows:

- How to specify the time interval during which messages received by a queue are not visible.

The scenario

In this example, a Node.js module is used to manage visibility timeout. The Node.js module uses the SDK for JavaScript to manage visibility timeout by using this method of the `SQS` client class:

- `ChangeMessageVisibilityCommand`

For more information about Amazon SQS visibility timeout, see Visibility timeout in the *Amazon Simple Queue Service 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 Loading credentials in Node.js from the shared credentials file (p. 33).
- Create an Amazon SQS queue. For an example of creating a queue, see Using queues in Amazon SQS (p. 268).
- Send a message to the queue. For an example of sending a message to a queue, see Sending and receiving messages in Amazon SQS (p. 273).

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 (p. 56).

Changing the visibility timeout

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

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

This example code can be found here on GitHub.

Create a Node.js module with the file name `sqs_changingvisibility.js`. Be sure to configure the SDK as previously shown. Receive the message from the queue.

Upon receipt of the message from the queue, create a JSON object containing the parameters needed for setting the timeout, including the URL of the queue containing the message, the `ReceiptHandle` returned when the message was received, and the new timeout in seconds. Call the `ChangeMessageVisibilityCommand` method.

Note
Replace and *`ACCOUNT_ID`* with the ID of the account, and *`QUEUE_NAME`* with the name of the queue.

```
// Import required AWS SDK clients and commands for Node.js
import {
  ReceiveMessageCommand,
  ChangeMessageVisibilityCommand,
```

```
} from  "@aws-sdk/client-sqs";
import { sqsClient } from  "./libs/sqsClient.js";

// Set the parameters
const queueURL = "https://sqs.REGION.amazonaws.com/ACCOUNT-ID/QUEUE-NAME"; // REGION,
 ACCOUNT_ID, QUEUE_NAME
const params = {
  AttributeNames: ["SentTimestamp"],
  MaxNumberOfMessages: 1,
  MessageAttributeNames: ["All"],
  QueueUrl: queueURL,
};


const run = async () => {
  try {
    const data = await sqsClient.send(new ReceiveMessageCommand(params));
    if (data.Messages != null) {
      try {
        var visibilityParams = {
          QueueUrl: queueURL,
          ReceiptHandle: data.Messages[0].ReceiptHandle,
          VisibilityTimeout: 20, // 20 second timeout
        };
        const results = await sqsClient.send(
          new ChangeMessageVisibilityCommand(params)
        );
        console.log("Timeout Changed", results);
      } catch (err) {
        console.log("Delete Error", err);
      }
    } else {
      console.log("No messages to change");
    }
    return data; // For unit tests.
  } catch (err) {
    console.log("Receive Error", err);
  }
};
run();
```

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

```
node sqs_changingvisibility.js
```

This example code can be found here on GitHub.

Enabling long polling in Amazon SQS

This Node.js code example shows:

- How to enable long polling for a newly created queue.
- How to enable long polling for an existing queue.
- How to enable long polling upon receipt of a message.

The scenario

Long polling reduces the number of empty responses by allowing Amazon SQS to wait a specified time for a message to become available in the queue before sending a response. Also, long polling eliminates false empty responses by querying all of the servers instead of a sampling of servers. To enable long polling, you must specify a non-zero wait time for received messages. You can do this by setting the `ReceiveMessageWaitTimeSeconds` parameter of a queue or by setting the `WaitTimeSeconds` parameter on a message when it is received.

In this example, a series of Node.js modules are used to enable long polling. The Node.js modules use the SDK for JavaScript to enable long polling using these methods of the `SQS` client class:

- `SetQueueAttributesCommand`
- `ReceiveMessageCommand`
- `CreateQueueCommand`

For more information about Amazon SQS long polling, see Long polling in the *Amazon Simple Queue Service 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.

> **Note**
> The AWS SDK for JavaScript (V3) is written in JavaScript, so for consistency these examples are presented in JavaScript. JavaScript is a super-set of JavaScript so these example can also be run in JavaScript.

- Create a shared configurations file with your user credentials. For more information about providing a shared credentials file, see Loading credentials in Node.js from the shared credentials file (p. 33).

> **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 (p. 56).

Enabling long polling when creating a queue

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

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

This example code can be found here on GitHub.

Create a Node.js module with the file name `sqs_longpolling_createqueue.js`. Be sure to configure the SDK as previously shown. Create a JSON object containing the parameters needed to create a queue, including a non-zero value for the `ReceiveMessageWaitTimeSeconds` parameter. Call the `CreateQueueCommand` method. Long polling is then enabled for the queue.

Note
Replace and *`SQS_QUEUE_URL`* with the URL of the SQS queue.

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

// Set the parameters
const params = {
  QueueName: "SQS_QUEUE_NAME", //SQS_QUEUE_URL; e.g., 'https://sqs.REGION.amazonaws.com/
ACCOUNT-ID/QUEUE-NAME'
  Attributes: {
    ReceiveMessageWaitTimeSeconds: "20",
  },
};

const run = async () => {
  try {
    const data = await sqsClient.send(new CreateQueueCommand(params));
    console.log("Success", data);
    return data; // For unit tests.
  } catch (err) {
    console.error(err, err.stack);
  }
};
run();
```

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

```
node sqs_longpolling_createqueue.js
```

This example code can be found here on GitHub.

Enabling long polling on an existing queue

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

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

This example code can be found here on GitHub.

Create a Node.js module with the file name `sqs_longpolling_existingqueue.js`. Be sure to configure the SDK as previously shown. Create a JSON object containing the parameters needed to set the attributes of the queue, including a non-zero value for the `ReceiveMessageWaitTimeSeconds` parameter and the URL of the queue. Call the `SetQueueAttributesCommand` method. Long polling is then enabled for the queue.

> **Note**
> Replace *SQS_QUEUE_URL* with the URL of the SQS queue, and *ReceiveMessageWaitTimeSeconds* with the number of seconds to wait before the message is received.

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

// Set the parameters
const params = {
  Attributes: {
    ReceiveMessageWaitTimeSeconds: "20",
  },
  QueueUrl: "SQS_QUEUE_URL", //SQS_QUEUE_URL; e.g., 'https://sqs.REGION.amazonaws.com/
ACCOUNT-ID/QUEUE-NAME'
};

const run = async () => {
  try {
    const data = await sqsClient.send(new SetQueueAttributesCommand(params));
    console.log("Success", data);
    return data; // For unit tests.
  } catch (err) {
    console.error(err, err.stack);
  }
};
run();
```

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

```
node sqs_longpolling_existingqueue.js
```

This example code can be found here on GitHub.

Enabling long polling on message receipt

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

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

This example code can be found here on GitHub.

Create a Node.js module with the file name `sqs_longpolling_receivemessage.js`. Be sure to configure the SDK as previously shown. Create a JSON object containing the parameters needed to receive messages, including a non-zero value for the `WaitTimeSeconds` parameter and the URL of the queue. Call the `ReceiveMessageCommand` method.

> **Note**
> Replace *SQS_QUEUE_URL* with the URL of the SQS queue, *MaxNumberOfMessages* with the maximum number of messages, and *WaitTimeSeconds* with the time to wait (in seconds).

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

// Set the parameters
const queueURL = "SQS_QUEUE_URL"; // SQS_QUEUE_URL
const params = {
  AttributeNames: ["SentTimestamp"],
  MaxNumberOfMessages: 1,
  MessageAttributeNames: ["All"],
  QueueUrl: queueURL,
  WaitTimeSeconds: 20,
};

const run = async () => {
  try {
    const data = await sqsClient.send(new ReceiveMessageCommand(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 sqs_longpolling_receivemessage.js
```

This example code can be found here on GitHub.

Using dead-letter queues in Amazon SQS

This Node.js code example shows:

- How to use a queue to receive and hold messages from other queues that the queues can't process.

The scenario

A dead-letter queue is one that other (source) queues can target for messages that can't be processed successfully. You can set aside and isolate these messages in the dead-letter queue to determine why their processing did not succeed. You must individually configure each source queue that sends messages to a dead-letter queue. Multiple queues can target a single dead-letter queue.

In this example, a Node.js module is used to route messages to a dead letter queue. The Node.js module uses the SDK for JavaScript to use dead letter queues using this method of the `SQS` client class:

- `SetQueueAttributesCommand`

For more information about Amazon SQS dead-letter queues, see Using Amazon SQS dead-letter queues in the *Amazon Simple Queue Service Developer Guide*.

Prerequisite tasks

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

- Create a shared configurations file with your user credentials. For more information about providing a shared credentials file, see Loading credentials in Node.js from the shared credentials file (p. 33).
- Create an Amazon SQS queue to serve as a dead-letter queue. For an example of creating a queue, see Using queues in Amazon SQS (p. 268).

> **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 (p. 56).

Configuring source queues

After you create a queue to act as a dead-letter queue, you must configure the other queues that route unprocessed messages to the dead-letter queue. To do this, specify a redrive policy that identifies the queue to use as a dead-letter queue and the maximum number of receives by individual messages before they are routed to the dead-letter queue.

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

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

This example code can be found here on GitHub.

Create a Node.js module with the file name `sqs_deadletterqueue.js`. Be sure to configure the SDK as previously shown, including downloading the required clients and packages. Create a JSON object containing the parameters needed to update queue attributes, including the `RedrivePolicy` parameter that specifies both the ARN of the dead-letter queue, as well as the value of `maxReceiveCount`. Also specify the URL source queue you want to configure. Call the `SetQueueAttributesCommand` method.

> **Note**
> Replace *`SQS_QUEUE_URL`* with the URL of the SQS queue, and *`DEAD_LETTER_QUEUE_ARN`* with the ARN of the dead letter queue.

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

// Set the parameters
var params = {
  Attributes: {
    RedrivePolicy:
      '{"deadLetterTargetArn":"DEAD_LETTER_QUEUE_ARN",' +
      '"maxReceiveCount":"10"}', //DEAD_LETTER_QUEUE_ARN
```

```
  },
  QueueUrl: "SQS_QUEUE_URL", //SQS_QUEUE_URL
};

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

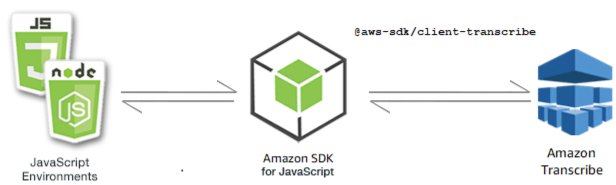
To run the example, enter the following at the command prompt.

```
node sqs_deadletterqueue.js // If you prefer JavaScript, enter 'sqs_deadletterqueue.js'
```

This example code can be found here on GitHub.

Amazon Transcribe examples

Amazon Transcribe makes it easy for developers to add speech to text capabilities to their applications.



The JavaScript API for Amazon Transcribe is exposed through the TranscribeService client class.

Topics

Amazon Transcribe examples

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 credentials JSON file, see Loading credentials in Node.js from the shared credentials file (p. 33).

> **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 (p. 56)

Starting an Amazon Transcribe job

This example demonstrates how to start a Amazon Transcribe transcription job using the AWS SDK for JavaScript. For more information, see StartTranscriptionJobCommand.

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

```
import  { TranscribeClient }  from  "@aws-sdk/client-transcribe";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create 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-job.js`. Make sure to configure the SDK as previously shown, including installing the required clients and packages. Create a parameters object, specifying the required parameters. Start the job using the `StartMedicalTranscriptionJobCommand` command.

> **Note**
> Replace *`MEDICAL_JOB_NAME`* with a name for the transcription job. For *`OUTPUT_BUCKET_NAME`* specify the Amazon S3 bucket where the output is saved. For *`JOB_TYPE`* specify types of job. For *`SOURCE_LOCATION`* specify the location of the source file. For *`SOURCE_FILE_LOCATION`* specify the location of the input media file.

```
// Import the required AWS SDK clients and commands for Node.js
import { StartTranscriptionJobCommand } from "@aws-sdk/client-transcribe";
import { transcribeClient } from "./libs/transcribeClient.js";

// Set the parameters
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"
  },
};

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

```
  try {
    const data = await transcribeClient.send(
      new StartTranscriptionJobCommand(params)
    );
    console.log("Success - put", data);
    return data; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

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

```
node transcribe-create-job.js
```

This sample code can be found here on GitHub.

List Amazon Transcribe jobs

This example shows how list the Amazon Transcribe transcription jobs using the AWS SDK for JavaScript. For more information about what other setting you can modify, see `ListTranscriptionJobCommand`.

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

```
import  { TranscribeClient }  from  "@aws-sdk/client-transcribe";
// Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"
// Create 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-list-jobs.js`. Make sure to configure the SDK as previously shown, including installing the required clients and packages. Create a parameters object with the required parameters.

> **Note**
> Replace ***`KEY_WORD`*** with a keyword that the returned jobs name must contain.

```
// Import the required AWS SDK clients and commands for Node.js

import { ListTranscriptionJobsCommand } from "@aws-sdk/client-transcribe";
import { transcribeClient } from "./libs/transcribeClient.js";

// Set the parameters
const params = {
  JobNameContains: "KEYWORD", // Not required. Returns only transcription
  // job names containing this string
};

const run = async () => {
  try {
    const data = await transcribeClient.send(
      new ListTranscriptionJobsCommand(params)
    );
```

```
    console.log("Success", data.TranscriptionJobSummaries);
    return data; // For unit tests.
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

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

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

This sample code can be found here on GitHub.

Deleting a Amazon Transcribe job

This example shows how to delete an Amazon Transcribe transcription job using the AWS SDK for JavaScript. For more information about optional, see `DeleteTranscriptionJobCommand`.

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

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

This example code can be found here on GitHub.

Create a Node.js module with the file name `transcribe-delete-job.js`. Make sure to configure the SDK as previously shown, including installing the required clients and packages. Specify the AWS Region, and the name of the job you want to delete.

> **Note**
> Replace *JOB_NAME* with the name of the job to delete.

```
// Import the required AWS SDK clients and commands for Node.js
import { DeleteTranscriptionJobCommand } from "@aws-sdk/client-transcribe";
import { transcribeClient } from "./libs/transcribeClient.js";

// Set the parameters
const params = {
  TranscriptionJobName: "JOB_NAME", // Required. For example, 'transciption_demo'
};

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

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

```
node transcribe-delete-job.js
```

This sample code can be found here on GitHub.

Amazon Transcribe medical examples

In this example, a series of Node.js modules are used to create,list, and delete medical transcription jobs using the following methods of the `TranscribeService` client class:

- `StartMedicalTranscriptionJobCommand`
- `ListMedicalTranscriptionJobsCommand`
- `DeleteMedicalTranscriptionJobCommand`

For more information about Amazon Transcribe users, see the Amazon Transcribe developer guide.

Prerequisite tasks

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

- Set up the project environment to run these Node TypeScript examples, and install the required AWS SDK for JavaScript and third-party modules. Follow the instructions on GitHub.
- Create a shared configurations file with your user credentials. For more information about providing a credentials JSON file, see Loading credentials in Node.js from the shared credentials file (p. 33).

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 (p. 56)

Starting an Amazon Transcribe medical transcription job

This example demonstrates how to start a Amazon Transcribe medical transcription job using the AWS SDK for JavaScript. For more information, see startMedicalTranscriptionJob.

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

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

This example code can be found here on GitHub.

Create a Node.js module with the file name `transcribe-create-medical-job.js`. Make sure to configure the SDK as previously shown, including installing the required clients and packages. Create a parameters object, specifying the required parameters. Start the medical job using the `StartMedicalTranscriptionJobCommand` command.

Note
Replace *MEDICAL_JOB_NAME* with a name for the medical transcription job. For *OUTPUT_BUCKET_NAME* specify the Amazon S3 bucket where the output is saved. For *JOB_TYPE* specify types of job. For *SOURCE_LOCATION* specify the location of the source file. For *SOURCE_FILE_LOCATION* specify the location of the input media file.

```
// Import the required AWS SDK clients and commands for Node.js
import { StartMedicalTranscriptionJobCommand } from "@aws-sdk/client-transcribe";
import { transcribeClient } from "./libs/transcribeClient.js";

// Set the parameters
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"
  },
};

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

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

```
node transcribe-create-medical-job.js
```

This sample code can be found here on GitHub.

Listing Amazon Transcribe medical jobs

This example shows how to list the Amazon Transcribe transcription jobs using the AWS SDK for JavaScript. For more information, see ListTranscriptionMedicalJobsCommand.

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

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

```
const REGION = "REGION"; //e.g. "us-east-1"
// Create 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-list-medical-jobs.js`. Make sure to configure the SDK as previously shown, including installing the required clients and packages. Create a paramters object with the required parameters, and list the medical jobs using the `ListMedicalTranscriptionJobsCommand` command.

Note
Replace *`KEYWORD`* with a keyword that the returned jobs name must contain.

```
// Import the required AWS SDK clients and commands for Node.js

import { ListMedicalTranscriptionJobsCommand } from "@aws-sdk/client-transcribe";
import { transcribeClient } from "./libs/transcribeClient.js";

// Set the parameters
const params = {
  JobNameContains: "KEYWORD", // Returns only transcription job names containing this
 string
};

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

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

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

This sample code can be found here on GitHub.

Deleting an Amazon Transcribe medical job

This example shows how to delete an Amazon Transcribe transcription job using the AWS SDK for JavaScript. For more information about optional, see `DeleteTranscriptionMedicalJobCommand`.

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

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

This example code can be found here on GitHub.

Create a Node.js module with the file name `transcribe-delete-job.js`. Make sure to configure the SDK as previously shown, including installing the required clients and packages. Create a parameters object with the required parameters, and delete the medical job using the `DeleteMedicalJobCommand` command.

> **Note**
> Replace *JOB_NAME* with the name of the job to delete.

```
// Import the required AWS SDK clients and commands for Node.js
import { DeleteMedicalTranscriptionJobCommand } from "@aws-sdk/client-transcribe";
import { transcribeClient } from "./libs/transcribeClient.js";

// Set the parameters
const params = {
  MedicalTranscriptionJobName: "MEDICAL_JOB_NAME", // For example,
 'medical_transciption_demo'
};

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();
```

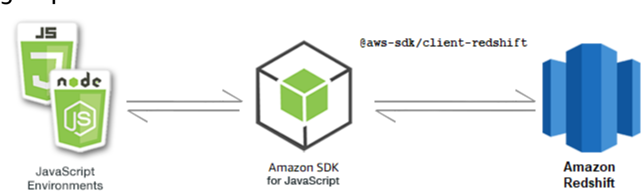
To run the example, enter the following at the command prompt.

```
node transcribe-delete-medical-job.js
```

This sample code can be found here on GitHub.

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 (p. 292)

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 credentials JSON file, see Loading credentials in Node.js from the shared credentials file (p. 33).

> **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 (p. 56)

Creating an Amazon Redshift cluster

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

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

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

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

This example code can be found here on GitHub.

Create a Node.js module with the file name `redshift-create-cluster.js`. Make sure to configure the SDK as previously shown, including installing the required clients and packages. Create a parameters object, specifying the node type to be provisioned, and the master username and password 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 master user name and password of the master user of your DB instance in the cluster. For *`CLUSTER_TYPE`*, enter the type of cluster. If you specify `single-node`, you do not require the `NumberOfNodes` parameter. The remaining parameters are optional.

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

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

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

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

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

This sample code can be found here on GitHub.

Modifing a Amazon Redshift cluster

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

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

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

This example code can be found here on GitHub.

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

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

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

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

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

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

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

This sample code can be found here on GitHub.

Viewing details of a Amazon Redshift cluster

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

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

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

This example code can be found here on GitHub.

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

> **Note**
> Replace *`CLUSTER_NAME`* with the name of the cluster.

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

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

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

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

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

This sample code can be found here on GitHub.

Delete an Amazon Redshift cluster

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

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

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

This example code can be found here on GitHub.

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

> **Note**
> Replace *`CLUSTER_NAME`* with the name of the cluster. For the *`SkipFinalClusterSnapshot`*, specify whether to create a final snapshot of the cluster before deleting it. If you specify 'false',

specify the id of the final cluster snapshot in *CLUSTER_SNAPSHOT_ID*. You can get this ID by clicking the link in the **Snapshots** column for the cluster on the **Clusters** dashboard, and scrolling down to the **Snapshots** pane. Note that the stem `rs:` is not part of the snapshot ID.

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

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

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

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

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

This sample code can be found here on GitHub.

Cross-service examples for the AWS SDK for JavaScript

The SDK for JavaScript enables you to use multiple AWS services in cooperation with each other to develop complex and sophisticated solutions. This section of the AWS SDK for JavaScript demonstrates several such solutions.

The following cross-service examples show you how to perform different tasks related to using the AWS SDK for JavaScript.

Topics

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*.

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.

   Warning
   AWS does not control the following code. Before you run it, be sure to verify its authenticity and integrity. More information about this code can be found in the nvm GitHub repository.

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.37.2/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.

```
. ~/.nvm/nvm.sh
```

4. Use nvm to install the latest version of Node.js by typing the following at the command line.

```
nvm install node
```

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. Once the Amazon EC2 instance goes away, you have to re-install Node again. The alternative is to make an Amazon Machine Image (AMI) of the Amazon EC2 instance once you have the configuration that you want to keep, as described in the following topic.

Creating an Amazon Machine Image (AMI)

After you install Node.js on an Amazon EC2 instance, you can create an Amazon Machine Image (AMI) from that instance. Creating an AMI makes it easy to provision multiple Amazon EC2 instances with the same Node.js installation. For more information about creating an AMI from an existing instance, see Creating an amazon EBS-backed Linux AMI in the *Amazon EC2 User Guide 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.

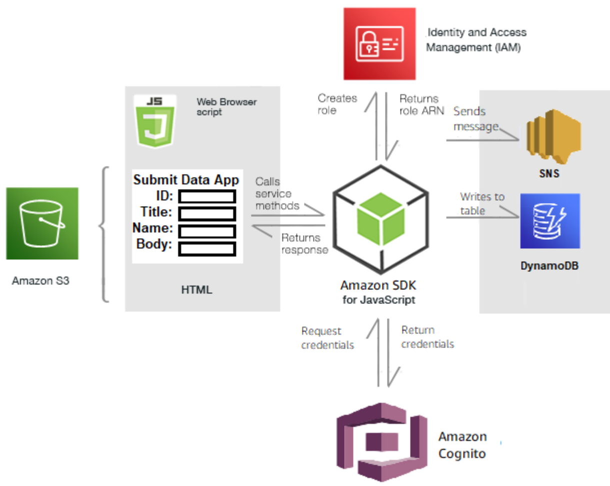
Build an app to submit data to DynamoDB

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 (p. 299)
2. Provision resources (p. 299)
3. Create the HTML (p. 301)
4. Create the browser script (p. 302)
5. Next steps (p. 304)

Prerequisites

Complete the following prerequisite tasks:

- Set up the project environment to run these Node TypeScript examples, and install the required AWS SDK for JavaScript and third-party modules. Follow the instructions on GitHub.
- Create a shared configurations file with your user credentials. For more information about providing a shared credentials file, see Loading credentials in Node.js from the shared credentials file (p. 33).

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 the AWS CloudFormation (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 developer 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 (CDK) Developer Guide.
3. Run the following command from the command line, replacing *STACK_NAME* with a unique name for the stack.

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

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

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

   To view the resources created, open AWS CloudFormation in the AWS management console, choose the stack, and select the **Resources** tab.
4. When the stack is create, use the AWS SDK for JavaScript to populate the DynamoDB table, as described in Populating the table (p. 300).

Populating the table

To populate the table, first create a diretory named `libs`, and in it create a file named `dynamoClient.js`, and paste the content below into it. Replace *REGION* with your AWS Region. This creates the DynamoDB client object.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
// Set the AWS Region.
const REGION = "REGION"; // e.g. "us-east-1"
// Create an Amazon DynamoDB service client object.
const dynamoClient = new DynamoDBClient({region:REGION});
export { dynamoClient };
```

This code is available here on GitHub.

Next, create a dynamoAppHelperFiles folder in your project folder, create a file update-table.js in it, and copy the content here on GitHub into it.

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

// Set the parameters.
export const params = {
  TableName: "Items",
  item: {
    Id: { N: "1" },
    Title: { S: "aTitle" },
    Name: { S: "aName" },
    Body: { S: "aBody" },
  },
};

export const run = async () => {
  try {
    const data = await dbclient.send(new PutItemCommand(params));
    console.log("success");
    console.log(data);
  } catch (err) {
    console.error(err);
  }
};
run();
```

Run the following command from the command line.

```
node update-table.js
```

This code is available here on GitHub.

Create a front-end page for the app

Here you create the front-end HTML browser page for the app.

Create a DynamoDBApp directory, create a file named index.html, and paste the code below into it. 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.

```
<!DOCTYPE html>
<html>
<head>
    <script type="text/javascript" src="./main.js"></script>
</head>
<body>
<h1>Submit an item to an Amazon DynamoDB Table</h1>
<p>Enter a value for each attribute and choose Submit </p>
<table style="width:100%">
    <tr>
```

```
        <td>ID:</td>
        <td><input type="text" id="id" name="id"/></td>
    </tr>
    <tr>
        <td>Title:</td>
        <td><input type="text" id="title" name="title"/></td>
    </tr>
    <tr>
        <td>Name:</td>
        <td><input type="text" id="name" name="name"/></td>
    </tr>
    <tr>
        <td>Body:</td>
        <td><input type="text" id="body" name="body"/></td>
    </tr>
        <td></td>
        <td><button type="button" onclick="submitData();">Submit</button></td>
        <script src="./main.js"></script>
    </tr>
</table>
</body>
</html>
```

This example code can be found here on GitHub.

Create the browser script

First, create the service client objects required for the example. Create a `libs` directory, create `snsClient.js`, and paste the code below into it. Replace *REGION* and *IDENTITY_POOL_ID* in each.

> **Note**
> Use the ID of the Amazon Cognito identity pool you created in Create the AWS resources (p. 299).

```
const { SNSClient } = require("@aws-sdk/client-sns");
const REGION = "REGION"; //e.g. "us-east-1"
const IdentityPoolId = "IDENTITY_POOL_ID";

const snsClient = new SNSClient({
  region: REGION,
  credentials: fromCognitoIdentityPool({
    client: new CognitoIdentityClient({ region: REGION }),
    identityPoolId: IdentityPoolId
  }),
});
export { snsClient };
```

This code is available here on GitHub..

To create the browser script for this example, in a folder named `DynamoDBApp`, create a Node.js module with the file name `add_dat.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. Replace *BUCKET_NAME* with the name of the S3 bucket you created. Next, create a parameters object for adding an item to the table. If none of the values is empty, `submitData` adds the item to the table, and sends the message. Remember to make the function available to the browser, with `window.submitData = submitData`

```
// Import required AWS SDK clients and commands for Node.js
```

```
import { PutItemCommand } from "@aws-sdk/client-dynamodb";
import { PublishCommand } from "@aws-sdk/client-sns";
import { snsClient } from "../libs/snsClient.js";
import { dynamoClient } from "../libs/dynamoClient.js";

export const submitData = async () => {
  //Set the parameters
  // Capture the values entered in each field in the browser (by id).
  const id = document.getElementById("id").value;
  const title = document.getElementById("title").value;
  const name = document.getElementById("name").value;
  const body = document.getElementById("body").value;
  //Set the table name.
  const tableName = "Items";

  //Set the parameters for the table
  const params = {
    TableName: tableName,
    // Define the attributes and values of the item to be added. Adding ' + "" ' converts a
 value to
    // a string.
    Item: {
      id: { N: id + "" },
      title: { S: title + "" },
      name: { S: name + "" },
      body: { S: body + "" },
    },
  };
  // Check that all the fields are completed.
  if (id != "" && title != "" && name != "" && body != "") {
    try {
      //Upload the item to the table
      const data = await dynamoClient.send(new PutItemCommand(params));
      alert("Data added to table.");
      try {
        // Create the message parameters object.
        const messageParams = {
          Message: "A new item with ID value was added to the DynamoDB",
          PhoneNumber: "+353861230764", //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`:

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

> **Note**
> For information about installing webpack, see Bundling applications with webpack (p. 42).

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 (p. 299) 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:

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 Loading credentials in Node.js from the shared credentials file (p. 33).

> **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 sytax, please refer to JavaScript ES6/CommonJS syntax (p. 56)

Create the AWS resources

This topic is part of a tutorial about building an app that transcribes and displays voice messages for authenticated users. To start at the beginning of the tutorial, see Build a transcription app with authenticated users (p. 304).

This topic desribes how to provison AWS resources for this app using the AWS Cloud Development Kit (CDK).

> **Note**
> 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 (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.

> **Note**
> 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 to store the transcriptions.

  Important
  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

This topic is part of a tutorial about building an app that transcribes and displays voice messages for authenticated users. To start at the beginning of the tutorial, see Build a transcription app with authenticated users (p. 304).

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.

```
<!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 id="demo" style="visibility: hidden;"></p>
</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">Transciption</td>
    <td style = "font-weight:bold">Download</td>
    <td style = "font-weight:bold">Delete</td>
       </tr>
</table>

<script type="text/javascript" src="./main.js"></script>
</body>

</html>
```

This code example is available here on GitHub.

Prepare the browser script

This topic is part of a tutorial about building an app that transcribes and displays voice messages for authenticated users. To start at the beginning of the tutorial, see Build a transcription app with authenticated users (p. 304).

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 topic describes in detail only the functions in `index.js` that use the SDK for JavaScript, which is available here on GitHub.

> **Note**
> `recorder.js` and `helper.js` are required but, because they do not contain Node.js code, are explained in the inline comments here and here 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 (p. 305) 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.

```
// 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";
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 COGNITO_ID = "COGNITO_ID";
// Get the Amazon Cognito ID token for the user. 'getToken()' is in 'helper.js'.
let idToken = getToken();
let loginData = {
  [COGNITO_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,
  credentials: fromCognitoIdentityPool({
    client: new CognitoIdentityClient({ region: params.Region }),
    identityPoolId: params.identityPoolID,
    logins: loginData,
  }),
});
```

When the HTML page loads, the `updateUserInterface` creates a folder with the user's name in the Amazon S3 bucket if its 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.

```
let updateUserInterface;
window.onload = updateUserInterface = 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(
          new ListObjectsCommand({ Bucket: params.Bucket, Prefix: username })
        );
        // 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 'yeidlUnit8Chunks' 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) {
        console.log("Error", err);
      }
    } catch (err) {
      console.log("Error creating presigned URL", err);
    }
  } catch (err) {
    console.log("Error", err);
  }
};

// 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.

```
// 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)}`;

  // 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.
    let signedUrl;
    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 {
    const data = 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);
  row.parentNode.removeChild(row);
};
```

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 (p. 42).

Run the app

This topic is part of a tutorial about building an app that transcribes and displays voice messages for authenticated users. To start at the beginning of the tutorial, see Build a transcription app with authenticated users (p. 304).

You can the 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

This topic is part of a tutorial about building an app that transcribes and displays voice messages for authenticated users. To start at the beginning of the tutorial, see Build a transcription app with authenticated users (p. 304).

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 an 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.

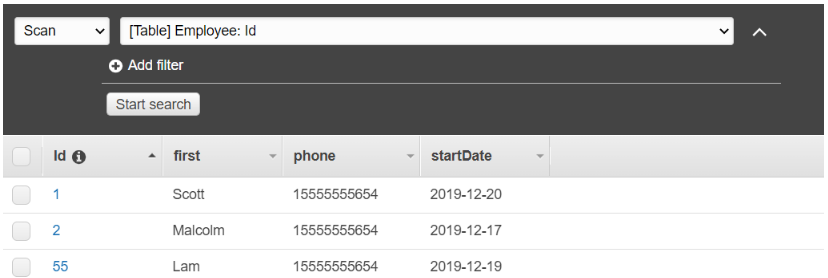
The example should take about 20 minutes to complete.

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 (p. 314)
2. Create the AWS resources (p. 314)
3. Prepare the browser script (p. 316)
4. Create and upload Lambda function (p. 316)
5. Deploy the Lambda function (p. 318)
6. Run the app (p. 319)
7. Delete the resources (p. 322)

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 (p. 312).

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 credentials JSON file, see Loading credentials in Node.js from the shared credentials file (p. 33).

> **Important**
> This example uses CommonJS sytax, as Lambda does not currently support ECMAScript 6 (ES6) syntax. For more information, see JavaScript ES6/CommonJS syntax (p. 56)

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 (p. 312).

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 developer 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 (CDK) Developer Guide.
3. Run the following command from the command line, replacing *STACK_NAME* with a unique name for the stack.

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

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

   For more information on the `create-stack` command parameters, see the AWS CLI Command Reference guide, and the AWS CloudFormation User Guide.
4. Next, populate the table by folllowing the procdure Populating the table (p. 314).

Populating the table

To populate the table, first create a diretory named `libs`, and in it create a file named `dynamoClient.js`, and paste the content below into it.

```
const { DynamoDBClient } = require ( "@aws-sdk/client-dynamodb" );
// Set the AWS Region.
```

```
const REGION = "REGION"; // e.g. "us-east-1"
 // Create an Amazon Lambda service client object.
const dynamoClient = new DynamoDBClient({region:REGION});
module.exports = { dynamoClient };
```

This code is available here on GitHub.

Next, create a file named `populate-table.js` in the root directory of your project folder, and copy the content here on GitHub into it. For one of the items, replace the value for the `phone` property with a valid mobile phone number in the E.164 format, and the value for the `startDate` with today's date.

Run the following command from the command line.

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

```
const { BatchWriteItemCommand } = require ( "aws-sdk/client-dynamodb" );
const {dynamoClient} = require ( "./libs/dynamoClient" );

// Set the parameters.
export const params = {
  RequestItems: {
    Employees: [
      {
        PutRequest: {
          Item: {
            id: { N: "1" },
            firstName: { S: "Bob" },
            phone: { N: "155555555555654" },
            startDate: { S: "2019-12-20" },
          },
        },
      },
      {
        PutRequest: {
          Item: {
            id: { N: "2" },
            firstName: { S: "Xing" },
            phone: { N: "155555555555653" },
            startDate: { S: "2019-12-17" },
          },
        },
      },
      {
        PutRequest: {
          Item: {
            id: { N: "55" },
            firstName: { S: "Harriette" },
            phone: { N: "155555555555652" },
            startDate: { S: "2019-12-19" },
          },
        },
      },
    ],
  },
};

export const run = async () => {
  try {
    const data = await dbclient.send(new BatchWriteItemCommand(params));
    console.log("Success", data);
  } catch (err) {
```

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

This code is available here on GitHub.

Creating the AWS Lambda function

Configuring the SDK

In the `libs` directory create a files named `snsClient.js` and `lambdaClient.js`, and paste the content below into them respectively.

```
const { SNSClient } = require("@aws-sdk/client-sns");
// Set the AWS Region.
const REGION = "REGION"; // e.g. "us-east-1"
// Create an Amazon Simple Notification Service client object.
const snsClient = new SNSClient({region:REGION});
module.exports = { snsClient };
```

Replace *REGION* with the AWS Region. This code is available here on GitHub.

```
const { LambadaClient } = require( "@aws-sdk/client-lambda" );
// Set the AWS Region.
const REGION = "eu-west-1"; // 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. Then create the parameters for the `ScanCommand`.Replace *TABLE_NAME* with the name of the table you created in the Create the AWS resources (p. 314) section of this example.

The following code snippet shows this step. (See Bundling the Lambda function (p. 317) for the full example.)

```
"use strict";
// Load the required clients and commands.
const { ScanCommand }  = require ( "@aws-sdk/client-dynamodb" );
const { PublishCommand } = require ( "@aws-sdk/client-sns" );
const {lambdaClient}  = require ( "./libs/lambdaClient" );
const {snsClient} = require ( "./libs/snsClient" );

// 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 anniversry 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 (p. 317) for the full example.)

```
exports.handler = async (event, context, callback) => {
  // Helper function to send message using Amazon SNS.
  async function sendText(textParams) {
    try {
      const data = await snsClient.send(new PublishCommand(textParams));
      console.log("Message sent");
    } catch (err) {
      console.log("Error, message not sent ", err);
    }
  }
  try {
    // Scan the table to check identify employees with work anniversary today.
    const data = await dynamoClient.send(new ScanCommand(params));
    data.Items.forEach(function (element, index, array) {
      const textParams = {
        PhoneNumber: element.phone.N,
        Message:
          "Hi " +
          element.firstName.S +
          "; congratulations on your work anniversary!",
      };
      // Send message using Amazon SNS.
      sendText(textParams);
    });
  } catch (err) {
    console.log("Error, could not scan table ", err);
  }
};
```

Bundling the Lambda function

This topic describes how to bundle the `mylambdafunction.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 (p. 313) for this example to install webpack.

   Note
   For information about*webpack*, see Bundling applications with webpack (p. 42).
2. Run the the following in the command line to bundle the JavaScript for this example into a file called `<index.js>` :

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

> **Important**
> Notice the output is named `index.js`. This is because Lambda functions must have an `index.js` handler to work.

3. Compress the bundled output file, `index.js`, into a ZIP file named `mylambdafunction.zip`.
4. Upload `mylambdafunction.zip` to the Amazon S3 bucket you created in the Create the AWS resources (p. 314) 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 (p. 312).

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 name 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 (p. 314) topic of this tutorial. Replace *LAMBDA_FUNCTION_NAME* with a name for the Lambda function.

```
// Load the required Lambda client and commands.
const {
  CreateFunctionCommand
} = require ( "@aws-sdk/client-lambda" );
const { lambdaClient} = require ( "./libs/lambdaClient.js );

// Set the parameters.
const params = {
  Code: {
    S3Bucket: "BUCKET_NAME", // BUCKET_NAME
    S3Key: "ZIP_FILE_NAME", // ZIP_FILE_NAME
  },
  FunctionName: "LAMBDA_FUNCTION_NAME",
  Handler: "index.handler",
  Role: "IAM_ROLE_ARN", // IAM_ROLE_ARN; e.g., arn:aws:iam::650138640062:role/v3-lambda-
tutorial-lambda-role
  Runtime: "nodejs12.x",
  Description:
    "Scans a DynamoDB table of employee details and using Amazon Simple Notification
 Services (Amazon SNS) to " +
    "send employees an email 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.

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

This code example is available here on GitHub.

Configure API Gateway to invoke the Lambda function

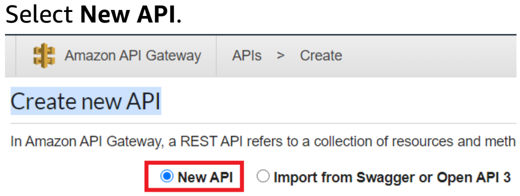
This topic is part of a tutorial that demonstrates how to invoke a Lambda function through Amazon API Gateway using the AWS SDK for JavaScript. To start at the beginning of the tutorial, see Invoking Lambda with API Gateway (p. 312).

To build the app:

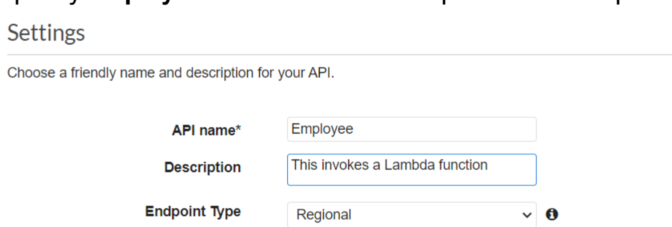
Create the rest API

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.

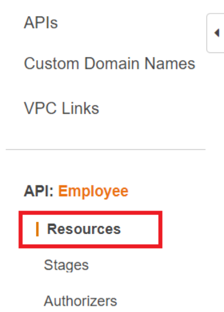
1. Sign in to the Amazon API Gateway console.
2. Under Rest API, choose **Build**.
3. Select **New API**.



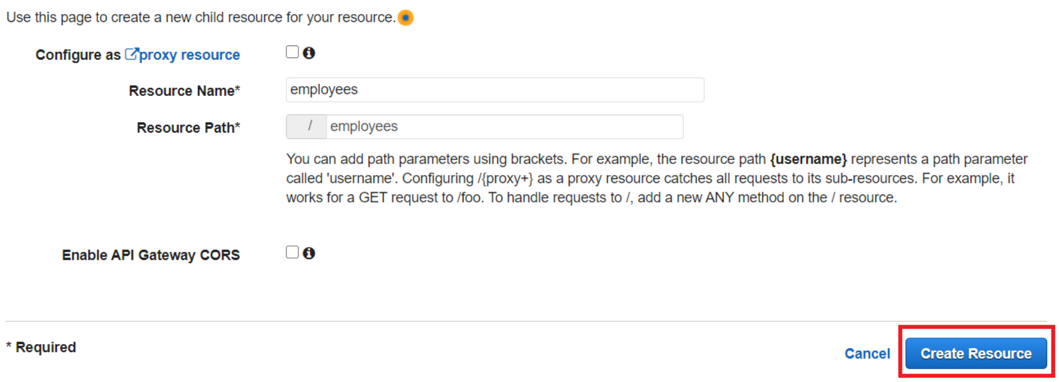
4. Specify **Employee** as the API name and provide a description.



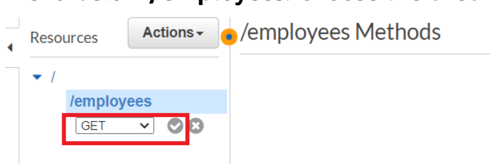
5. Choose **Create API**.
6. Choose **Resources** under the **Employee** section.



7. In the name field, specify **employees**.
8. Choose **Create Resources**.
9. From the **Actions** dropdown, choose **Create Resources**.



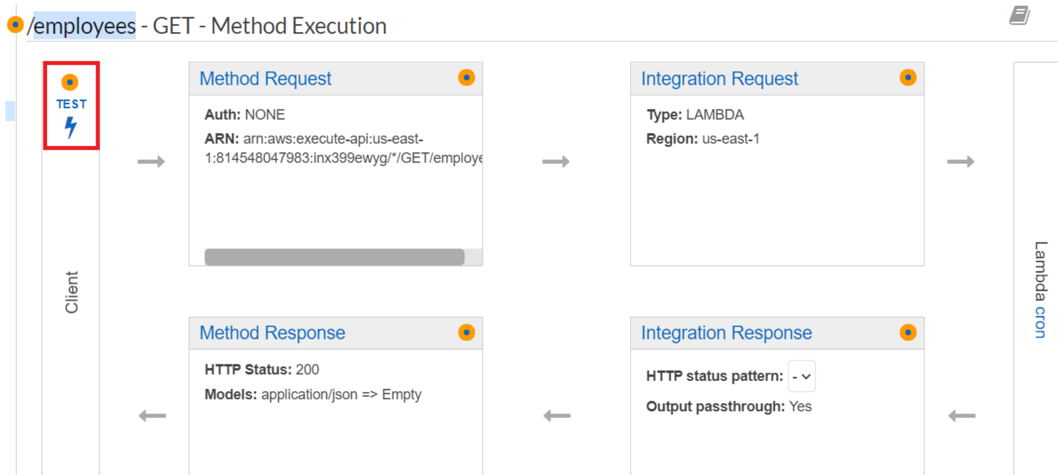
10. Choose **/employees**, select **Create Method** from the **Actions**, then select **GET** from the drop-down menu below **/employees**. Choose the checkmark icon.



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.

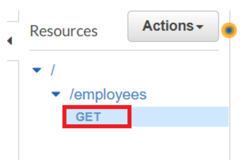


Once the Lambda function is invoked, you can view the log file to see a successful message.

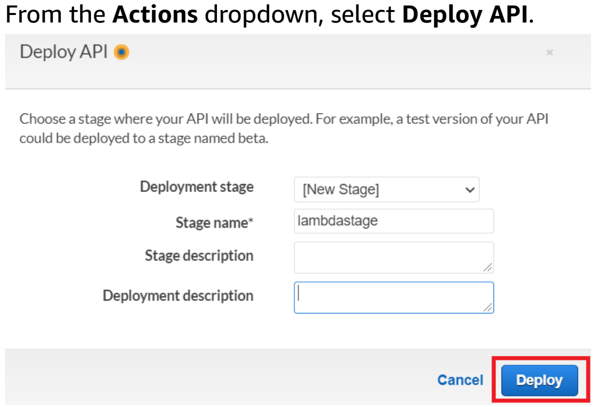
Deploy the API Gateway method

After the test is successful, you can deploy the method from the Amazon API Gateway console.

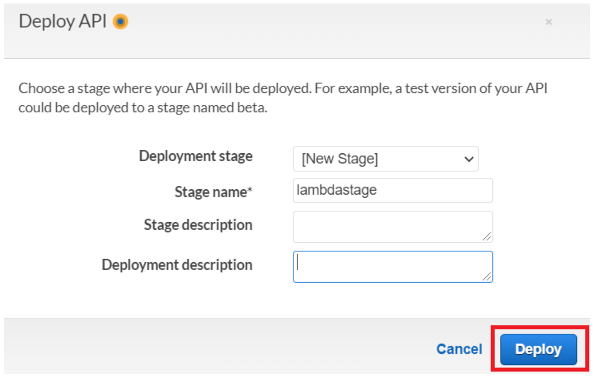
1. Choose **Get**.



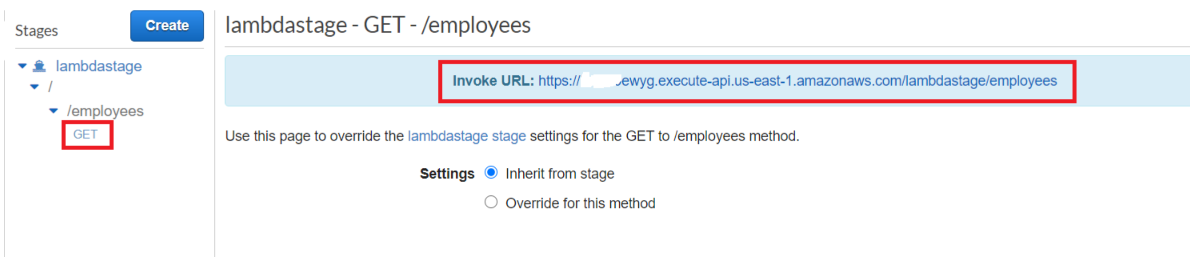
2. From the **Actions** dropdown, select **Deploy API**.



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 (p. 312).

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 (p. 314) 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 AWS 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:

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 (p. 322).

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 credentials JSON file, see Loading credentials in Node.js from the shared credentials file (p. 33).

> **Important**
> This example uses CommonJS sytax, as Lambda does not currently support ECMAScript 6 (ES6) syntax. For more information, see JavaScript ES6/CommonJS syntax (p. 56)

Create the AWS resources

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 (p. 322).

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 (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 developer 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 (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.

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.

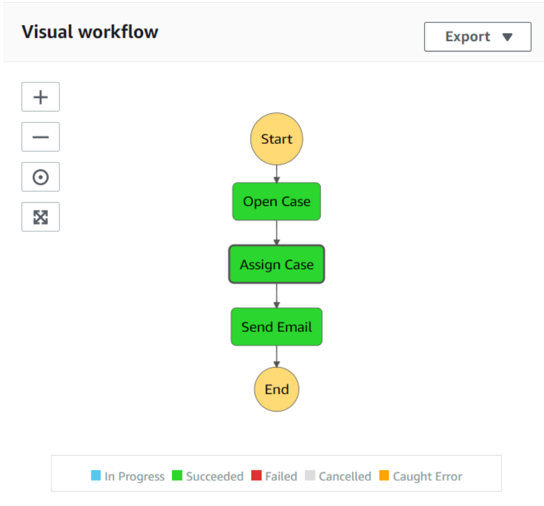
Important
The stack name must be unique within an AWS Region and AWS account. You can specify up to 128 characters, and numbers and hyphens are allowed.

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 (p. 322).

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.

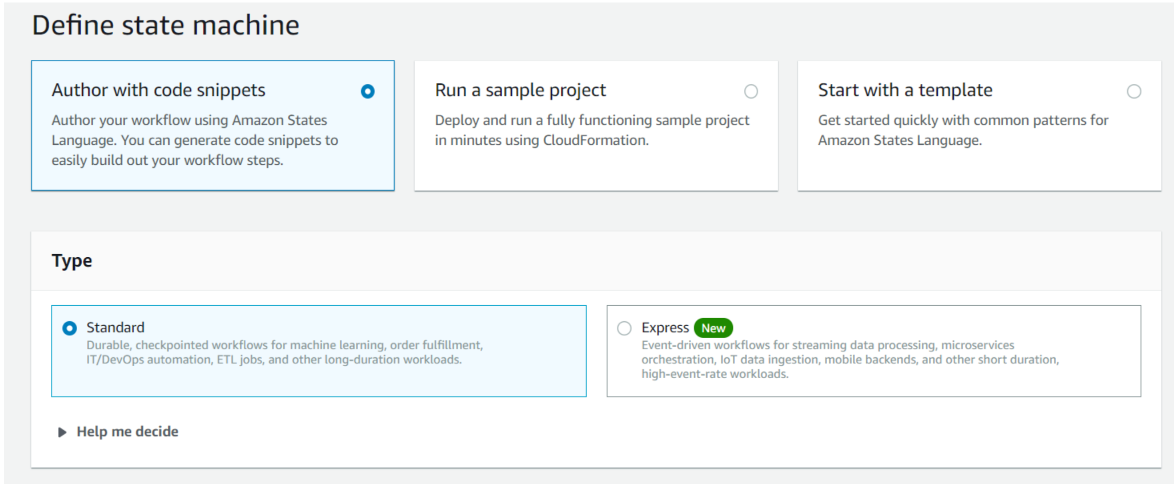
Create a serverless workflow by using Step functions

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**.



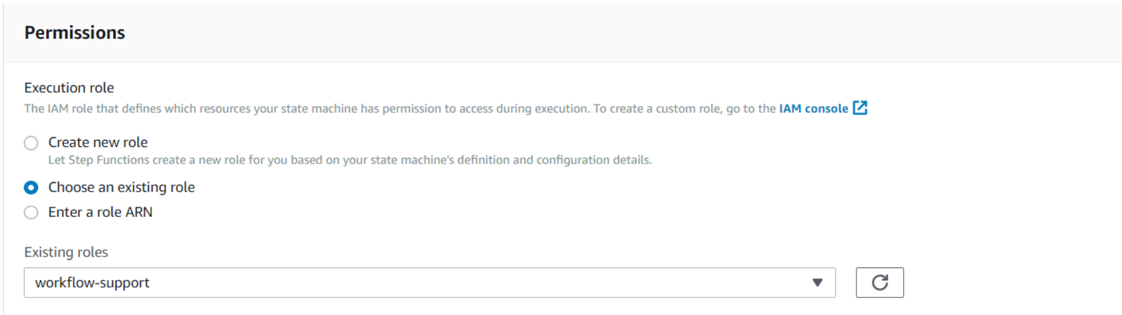
4. Specify the Amazon States Language document by entering the following code.

```
{
"Comment": "A simple AWS Step Functions state machine that automates a call center
 support session.",
"StartAt": "Open Case",
"States": {
"Open Case": {
"Type": "Task",
"Resource": "arn:aws:lambda:REGION:ACCOUNT_ID:function:FUNCTION_NAME",
"Next": "Assign Case"
},
"Assign Case": {
"Type": "Task",
"Resource": "arn:aws:lambda:REGION:ACCOUNT_ID:function:FUNCTION_NAME",
"Next": "Send Email"
},
"Send Email": {
"Type": "Task",
"Resource": "arn:aws:lambda:REGION:ACCOUNT_ID:function:FUNCTION_NAME",
"End": true
}
}
}
```

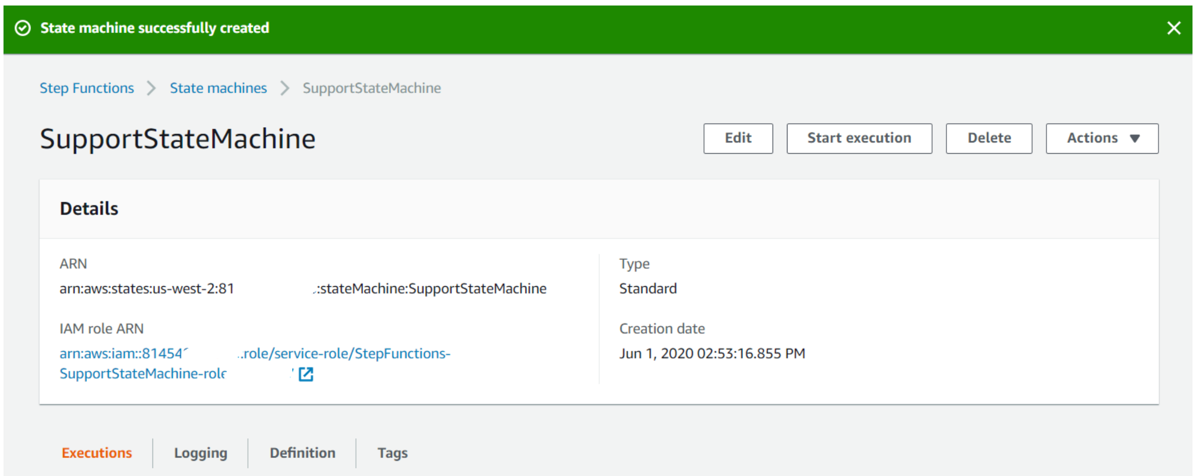
Note

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 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 (p. 322).

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 (p. 327) - Used as the first step in the workflow that processes the ticket ID value.
- addItem Lambda class (p. 328) - 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 (p. 329) - Used as the third step in the workflow that use 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.

```
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 { DynamoDBClient, PutItemCommand } = require("@aws-sdk/client-dynamodb");

const REGION = "eu-west-1"; //e.g. "us-east-1"
// Create the client service objects.
    const dbclient = new DynamoDBClient({ region: REGION });
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 = {
                TableName: "Case",
                Item: {
                    id: {N: val.Case},
                    empEmail: {S: "brmur@amazon.com"},
                    name: {S: "Tom Blue"}
                },
            }
            console.log('adding item for tom');
            try {
                const data = await dbclient.send(new PutItemCommand(params));
                console.log(data);
            } catch (err) {
                console.error(err);
            }
            var result = { Email: params.Item.empEmail };
            return result;
        } else {
            const params = {
```

```
                TableName: "Case",
                Item: {
                    id: {N: val.Case},
                    empEmail: {S: "brmur@amazon.com"},
                    name: {S: "Sarah White"}
                },
            }
            console.log('adding item for sarah');
            try {
                const data = await dbclient.send(new PutItemCommand(params));
                console.log(data);
            } catch (err) {
                console.error(err);
            }
            return params.Item.empEmail;
            var result = { Email: params.Item.empEmail };
        }
    }
    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 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 the topic of this example.

This code example is available here on GitHub.

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.

```

// Load the required clients and commands.
const { SESClient, SendEmailCommand } = require("@aws-sdk/client-ses");

// Set the AWS Region.
const REGION = "eu-west-1"; //e.g. "us-east-1"

// Create the client service objects.
const sesclient = new SESClient({ region: REGION });

exports.handler = async (event) => {
    // Enter a sender email address. This address must be verified.
    const sender = "Sender Name <briangermurray@gmail.com>";

    // 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,\r\n"
        + "Please check the database for new ticket assigned to you.";

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

// The character encoding for the email.
    const charset = "UTF-8";
    var params = {
        Source: sender,
        Destination: {
            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`.

```
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.

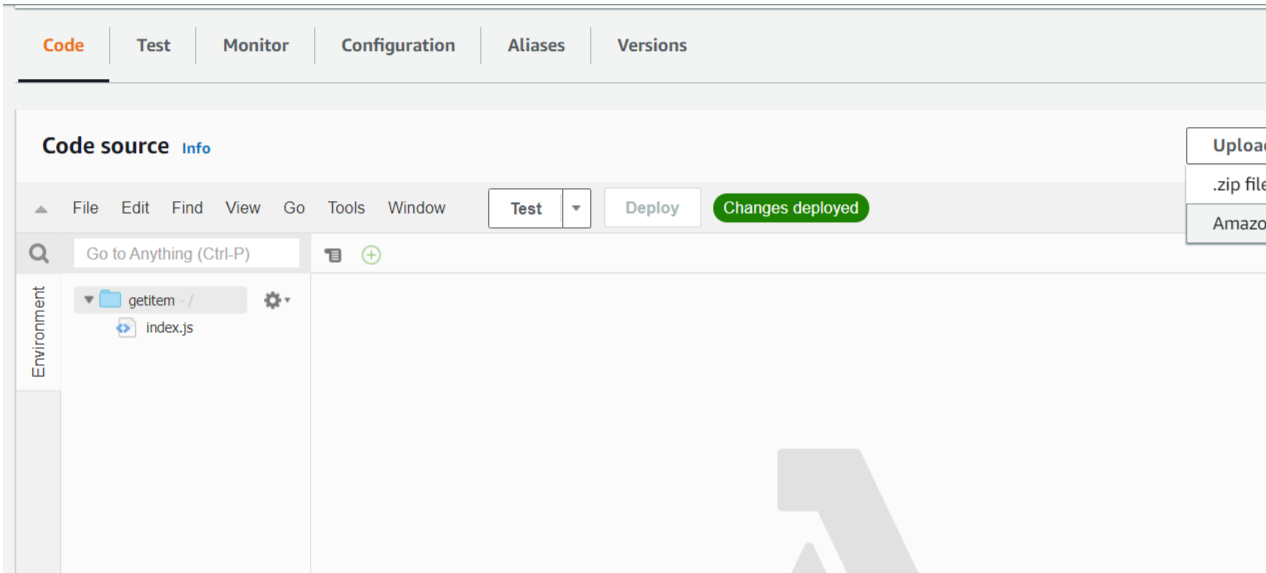
This code example is available here on GitHub.

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**.

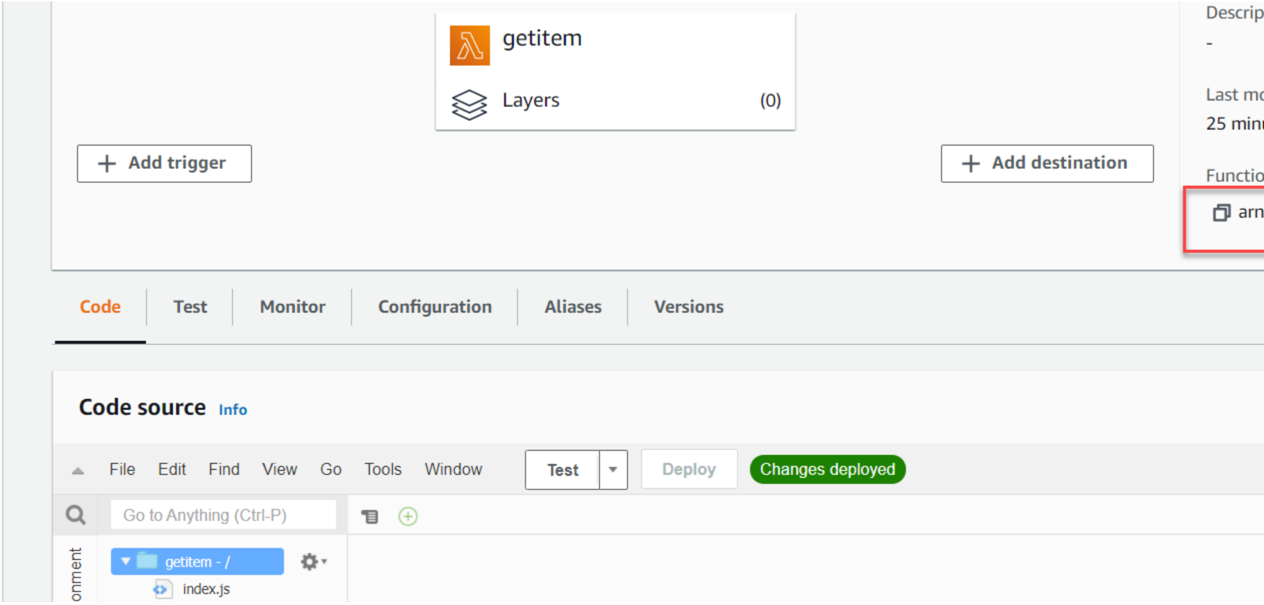


10. Choose **Save.**
11. Repeat this procedure for the **additem.js.zip** and **sendemail.js.zip** to new Lambda functions. When you finish, you will have three Lambda functions that you can reference in the Amazon States Language document.

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 (p. 322).

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

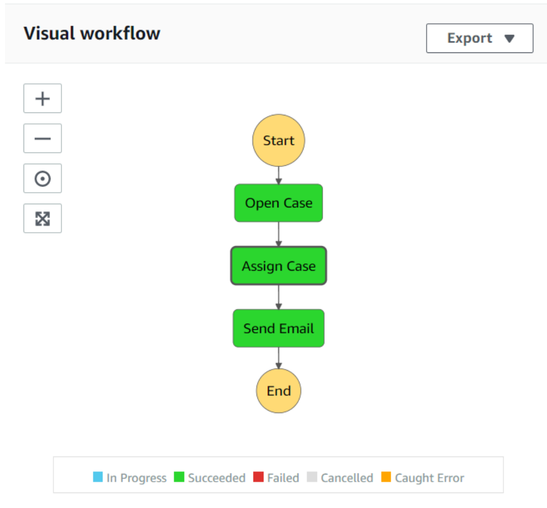
This topic is part of a tutorial that demonstrates to to invoke Lambda functions using AWS Step Functions. To start at the beginning of the tutorial, see Creating AWS serverless workflows using AWS SDK for JavaScript (p. 322).

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.

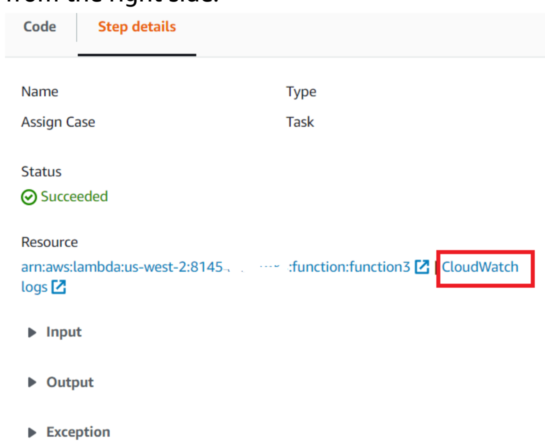
```
{
"inputCaseID": "001"
}
```

To execute your workflow:

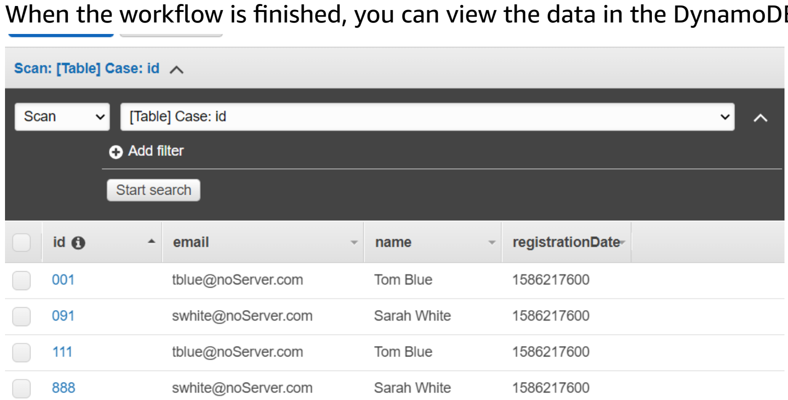
1. On the Step Functions console, choose **Start execution**.
2. In the **Input** section, pass the JSON data. View the workflow. As each step is completed, it turns green.



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 (p. 323) 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 an Lambda function every weekday.

AWS Lambda is a compute service that enables you to run code without provisioning or managing servers. You can create Lambda functions in various programming languages. For more information about AWS Lambda, see What is AWS Lambda.

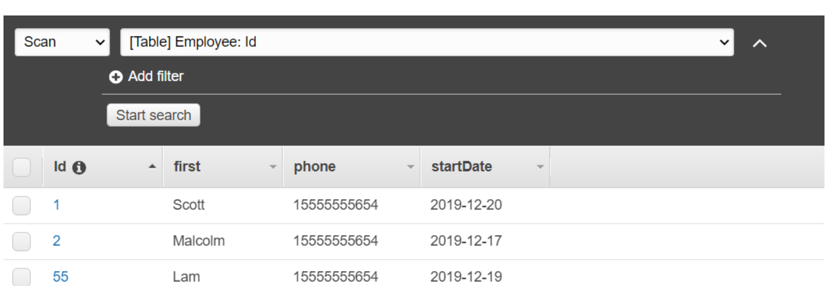
In this tutorial, you create a Lambda function by using the Lambda JavaScript runtime API. This example invokes different AWS services to perform a specific use case. For example, assume that an organization sends a mobile text message to its employees that congratulates them at the one year anniversary date, as shown in this illustration.

The tutorial should take about 20 minutes to complete.

This tutorial shows you how to use JavaScript logic to create a solution that performs this use case. For example, you'll learn how to read a database to determine which employees have reached the one year anniversary date, how to process the data, and send out a text message all by using a Lambda function. Then you'll learn how to use a cron expression to invoke the Lambda function every weekday.

This AWS tutorial uses an Amazon DynamoDB table named Employee that contains these fields.

- **id** - the primary key for the table.
- **firstName** - employee's first name.
- **phone** - employee's phone number.
- **startDate** - employee's start date.



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 (p. 335)
2. Create the AWS resources (p. 335)
3. Prepare the browser script (p. 337)
4. Create and upload Lambda function (p. 337)
5. Deploy the Lambda function (p. 340)
6. Run the app (p. 341)
7. Delete the resources (p. 341)

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 (p. 334).

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 credentials JSON file, see Loading credentials in Node.js from the shared credentials file (p. 33).

Important
This example uses CommonJS sytax, as Lambda does not currently support ECMAScript 6 (ES6) syntax. For more information, see JavaScript ES6/CommonJS syntax (p. 56)

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 (p. 334).

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 developer 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 (CDK) Developer Guide.
3. Run the following command from the command line, replacing *`STACK_NAME`* with a unique name for the stack.

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

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

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

   View a list of the resources in the console by opening the stack on the AWS CloudFormation dashboard, and choosing the **Resources** tab. You require these for the tutorial.
4. When the stack is created, use the AWS SDK for JavaScript to populate the DynamoDB table, as described in Populate the DynamoDB table (p. 336).

Populate the DynamoDB table

To populate the table, first create a diretory named `libs`, and in it create a file named `dynamoClient.js`, and paste the content below into it.

```
const { DynamoDBClient } = require( "@aws-sdk/client-dynamodb" );
// Set the AWS Region.
const REGION = "REGION"; // e.g. "us-east-1"
// Create an Amazon DynamodDB service client object.
const dynamoClient = new DynamoDBClient({region:REGION});
module.exports = { dynamoClient };
```

This code is available here on GitHub.

Next, create a file named `populate-table.js` in the root directory of your project folder, and copy the content here on GitHub into it. For one of the items, replace the value for the `phone` property with a valid mobile phone number in the E.164 format, and the value for the `startDate` with today's date.

Run the following command from the command line.

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

```
const {
BatchWriteItemCommand } = require( "aws-sdk/client-dynamodb" );
const {dynamoClient} = require(  "./libs/dynamoClient" );
// Set the parameters.
const params = {
  RequestItems: {
    Employees: [
      {
        PutRequest: {
          Item: {
            id: { N: "1" },
            firstName: { S: "Bob" },
            phone: { N: "155555555555654" },
            startDate: { S: "2019-12-20" },
          },
        },
      },
      {
        PutRequest: {
          Item: {
            id: { N: "2" },
            firstName: { S: "Xing" },
            phone: { N: "155555555555653" },
            startDate: { S: "2019-12-17" },
          },
        },
      },
      {
        PutRequest: {
          Item: {
            id: { N: "55" },
            firstName: { S: "Harriette" },
            phone: { N: "155555555555652" },
            startDate: { S: "2019-12-19" },
          },
        },
      },
    ],
  },
};

export const run = async () => {
  try {
    const data = await dbclient.send(new BatchWriteItemCommand(params));
    console.log("Success", data);
  } catch (err) {
    console.log("Error", err);
  }
};
run();
```

This code is available here on GitHub.

Creating the AWS Lambda function

Configuring the SDK

First import the required AWS SDK for JavaScript (v3) modules and commands: `DynamoDBClient` and the DynamoDB `ScanCommand`, and `SNSClient` and the Amazon SNS `PublishCommand` command. Replace *`REGION`* with the AWS Region. Then calculate today's date and assign it to a parameter. Then create the parameters for the `ScanCommand`.Replace *`TABLE_NAME`* with the name of the table you created in the Create the AWS resources (p. 335) section of this example.

The following code snippet shows this step. (See Bundling the Lambda function (p. 339) for the full example.)

```
"use strict";
// Load the required clients and commands.
const { DynamoDBClient, ScanCommand } = require("@aws-sdk/client-dynamodb");
const { SNSClient, PublishCommand } = require("@aws-sdk/client-sns");

//Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"

// Get today's date.
const today = new Date();
const dd = String(today.getDate()).padStart(2, "0");
const mm = String(today.getMonth() + 1).padStart(2, "0"); //January is 0!
const yyyy = today.getFullYear();
const date = yyyy + "-" + mm + "-" + dd;

// Set the parameters for the ScanCommand method.
const params = {
  // Specify which items in the results are returned.
  FilterExpression: "startDate = :topic",
  // Define the expression attribute value, which are substitutes for the values you want
 to compare.
  ExpressionAttributeValues: {
    ":topic": { S: date },
  },
  // Set the projection expression, which the the attributes that you want.
  ProjectionExpression: "firstName, phone",
  TableName: "TABLE_NAME",
};
```

Scanning the DynamoDB table

First create an async/await function called `sendText` to publish a text message using the Amazon SNS `PublishCommand`. Then, add a `try` block pattern that scans the DynamoDB table for employees with their work anniversry 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 (p. 339) for the full example.)

```
exports.handler = async (event, context, callback) => {
  // Helper function to send message using Amazon SNS.
  async function sendText(textParams) {
    try {
      const data = await snsclient.send(new PublishCommand(textParams));
      console.log("Message sent");
    } catch (err) {
      console.log("Error, message not sent ", err);
    }
  }
  try {
    // Scan the table to check identify employees with work anniversary today.
    const data = await dbclient.send(new ScanCommand(params));
    data.Items.forEach(function (element, index, array) {
      const textParams = {
        PhoneNumber: element.phone.N,
        Message:
          "Hi " +
          element.firstName.S +
          "; congratulations on your work anniversary!",
```

```
      };
      // Send message using Amazon SNS.
      sendText(textParams);
    });
  } catch (err) {
    console.log("Error, could not scan table ", err);
  }
};
```

Bundling the Lambda function

This topic describes how to bundle the `mylambdafunction.js` and the required AWS SDK for JavaScript modules for this example into a bundled file called `index.js`.

1. If you haven't already, follow the Prerequisite tasks (p. 335) for this example to install webpack.

   Note
   For information about*webpack*, see Bundling applications with webpack (p. 42).
2. Run the the following in the command line to bundle the JavaScript for this example into a file called `<index.js>` :

```
webpack mylamdbafunction.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 (p. 335) topic of this tutorial.

Here is the complete browser script code for `mylambdafunction.js`.

```
"use strict";
// Load the required clients and commands.
const { DynamoDBClient, ScanCommand } = require("@aws-sdk/client-dynamodb");
const { SNSClient, PublishCommand } = require("@aws-sdk/client-sns");

//Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"

// Get today's date.
const today = new Date();
const dd = String(today.getDate()).padStart(2, "0");
const mm = String(today.getMonth() + 1).padStart(2, "0"); //January is 0!
const yyyy = today.getFullYear();
const date = yyyy + "-" + mm + "-" + dd;

// Set the parameters for the ScanCommand method.
const params = {
  // Specify which items in the results are returned.
  FilterExpression: "startDate = :topic",
  // Define the expression attribute value, which are substitutes for the values you want
 to compare.
  ExpressionAttributeValues: {
    ":topic": { S: date },
  },
  // Set the projection expression, which the the attributes that you want.
  ProjectionExpression: "firstName, phone",
```

```
  TableName: "TABLE_NAME",
};

// Create the client service objects.
const dbclient = new DynamoDBClient({ region: REGION });
const snsclient = new SNSClient({ region: REGION });

exports.handler = async (event, context, callback) => {
  // Helper function to send message using Amazon SNS.
  async function sendText(textParams) {
    try {
      const data = await snsclient.send(new PublishCommand(textParams));
      console.log("Message sent");
    } catch (err) {
      console.log("Error, message not sent ", err);
    }
  }
  try {
    // Scan the table to check identify employees with work anniversary today.
    const data = await dbclient.send(new ScanCommand(params));
    data.Items.forEach(function (element, index, array) {
      const textParams = {
        PhoneNumber: element.phone.N,
        Message:
          "Hi " +
          element.firstName.S +
          "; congratulations on your work anniversary!",
      };
      // Send message using Amazon SNS.
      sendText(textParams);
    });
  } catch (err) {
    console.log("Error, could not scan table ", err);
  }
};
```

Deploy the Lambda function

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 (p. 334).

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 (p. 335) topic of this tutorial. Replace *`LAMBDA_FUNCTION_NAME`* with a name for the Lambda function.

```
// Load the required Lambda client and commands.
const {
   CreateFunctionCommand,
} = require("@aws-sdk/client-lambda");
const {
   lambdaClient
} = require("..libs/lambdaClient.js");

// Instantiate an Lambda client service object.
const lambda = new LambdaClient({ region: REGION });
```

```
// Set the parameters.
const params = {
  Code: {
    S3Bucket: "BUCKET_NAME", // BUCKET_NAME
    S3Key: "ZIP_FILE_NAME", // ZIP_FILE_NAME
  },
  FunctionName: "LAMBDA_FUNCTION_NAME",
  Handler: "index.handler",
  Role: "IAM_ROLE_ARN", // IAM_ROLE_ARN; e.g., arn:aws:iam::650138640062:role/v3-lambda-
tutorial-lambda-role
  Runtime: "nodejs12.x",
  Description:
    "Scans a DynamoDB table of employee details and using Amazon Simple Notification
 Services (Amazon SNS) to " +
    "send employees an email the each anniversary of their start-date.",
};

const run = async () => {
  try {
    const data = await lambda.send(new CreateFunctionCommand(params));
    console.log("Success", data); // successful response
  } catch (err) {
    console.log("Error", err); // an error occurred
  }
};
run();
```

Enter the following at the command line to deploy the Lambda function.

```
node lambda-function-setup.js
```

This code example is available here on GitHub.

Configure CloudWatch to invoke the Lambda functions

To configure CloudWatch to invoke the Lambda functions:

1. Open the **Functions** page on the Lambda console.
2. Choose the Lambda function.
3. Under **Designer**, choose **Add trigger**.
4. Set the trigger type to **CloudWatch Events/EventBridge**.
5. For Rule, choose **Create a new rule**.
6. Fill in the Rule name and Rule description.
7. For rule type, select **Schedule expression**.
8. In the **Schedule expression** field, enter a cron expression. For example, **cron(0 12 ? * MON-FRI *)**.
9. Choose **Add**.

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

Delete the resources

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 (p. 334).

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 (p. 335) 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 and using Lambda functions

The tutorial describes how to create and execute from the browser a Lambda function that creates a DynamoDB table.

The tutorial should take about 20 minutes to complete.

To build the app:

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 credentials JSON file, see Loading credentials in Node.js from the shared credentials file (p. 33).

Important
This example uses CommonJS sytax, as Lambda does not currently support ECMAScript 6 (ES6) syntax. For more information, see JavaScript ES6/CommonJS syntax (p. 56)

Create the AWS resources

This topic is part of a tutorial that demonstrates how to create, deploy, and run a Lambda function using the AWS SDK for JavaScript. To start at the beginning of the tutorial, see Creating and using Lambda functions (p. 342).

This tutorial requires the following resources.

- An Amazon Cognito identity pool with an unauthenticated user role.

- An IAM policy with permissions for the DynamoDB and Lambda is attached to the unauthenticated user role.
- An Amazon S3 bucket to host the browser HTML and script pages, and the Lambda function.

You can create these resources manually, but we recommend provisioning these resources using the AWS Cloud Development Kit (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 developer 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 `describe-stack-resources.js` in the root directory of your project folder.
3. 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 (CDK) Developer Guide.
4. 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.
5. Copy and paste the code below into `describe-stack-resources.js`.

```
// Load the AWS SDK for Node.js
const {
  CloudFormationClient,
  DescribeStackResourcesCommand,
  CreateStackCommand,
  DescribeStacksCommand
} = require("@aws-sdk/client-cloudformation");

// Create S3 service object
const cloudformation = new CloudFormationClient();

var params = {
  StackName: process.argv[2]
}

const getVariables = async () => {
      try {
        const data = await cloudformation.send(
            new DescribeStacksCommand({StackName: params.StackName}));
```

```
        console.log('Status: ', data.Stacks[0].StackStatus);
        if (data.Stacks[0].StackStatus == "CREATE_COMPLETE") {
          const data = await cloudformation.send(
              new DescribeStackResourcesCommand({StackName: params.StackName})
          );
          for (var i = 0; i < data.StackResources.length; i++) {
            var obj = data.StackResources[i].ResourceType;
            if (obj == "AWS::IAM::Policy") {
              const IDENTITY_POOL_ID = data.StackResources[i].LogicalResourceId;
              console.log("IDENTITY_POOL_ID:", IDENTITY_POOL_ID);
              var identity_pool_id = IDENTITY_POOL_ID;
            }
            if (obj == "AWS::S3::Bucket") {
              const BUCKET_NAME = data.StackResources[i].PhysicalResourceId;
              console.log("BUCKET_NAME:", BUCKET_NAME);
              var bucket = BUCKET_NAME;
            }
            if (obj == "AWS::IAM::Role") {
              const IAM_ROLE = data.StackResources[i].StackId;
              console.log("IAM_ROLE:", IAM_ROLE);
              var iam_role = IAM_ROLE;
            }
          }
        }
        else{
          console.log('Stack not ready yet. Try again in a few minutes.')
        }
      }catch (err) {
        console.log("Error listing resources", err);
      }
    }
;
getVariables();
```

This code is available here on GitHub.

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

```
node describe-stack-resources.js STACK_NAME
```

Take note of the *IAM_ROLE*, *IDENTITY_POOL_ID*, and *BUCKET_NAME* returned in the command line, as you need them for this tutorial.

Create the HTML

This topic is part of a tutorial that demonstrates how to create, deploy, and run a Lambda function using the AWS SDK for JavaScript. To start at the beginning of the tutorial, see Creating and using Lambda functions (p. 342).

First, create a `LambdaApp` folder. In this folder, create an `index.html` file, and copy and paste the content below into it. Upload the `index.html` file to the Amazon S3 bucket you created in the Create the AWS resources (p. 342) topic of this tutorial.

```
<!doctype html>
<html>
<head>
<meta charset="UTF-8">
<title>Card Slots</title>
```

```
</head>

<body>
    <button type="button" onclick="createtable()">Create a table!</button>
 <script type="text/javascript" src="main.js"></script>
</body>
</html>
```

This code example is available here on GitHub.

Prepare the browser script

This topic is part of a tutorial that demonstrates how to create, deploy, and run a Lambda function using the AWS SDK for JavaScript. To start at the beginning of the tutorial, see Creating and using Lambda functions (p. 342).

First, create the required service client objects. Create a `libs` folder, and in it create two files, `dynamoClient.js` and `lambdaClient.js`. Paste the code below into `dynamoClient.js`.

```
const { CognitoIdentityClient } = require ( "@aws-sdk/client-cognito-identity" );
const { fromCognitoIdentityPool } = require ( "@aws-sdk/credential-provider-cognito-
identity" );
const { DynamoDBClient } = require ( "@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,
  }),
});

module.exports = { dynamoClient };
```

Paste the code below into `lambdaClient.js`.

```
const { lambdaClient } = require ("@aws-sdk/client-lambda" );
const {
  fromCognitoIdentityPool,
} = require ( "@aws-sdk/credential-provider-cognito-identity" );
const { CognitoIdentityClient }  = require ("@aws-sdk/client-cognito-identity" );

// Set the AWS Region.
const REGION = "eu-west-1"; // e.g., 'us-east-2'
const IDENTITY_POOL_ID = "eu-west-1:dc7d706a-1f07-4fa5-baa7-edfabc05f293";

// Create an AWS Lambda client service object that initializes the Amazon Cognito
 credentials provider.
const lambdaClient = new LambdaClient({
  region: REGION,
  credentials: fromCognitoIdentityPool({
    client: new CognitoIdentityClient({ region: REGION }),
    identityPoolId: IDENTITY_POOL_ID
  }),
});
module.exports = {lambdaClient}
```

In both, replace *REGION* with the AWS region. Create an Lambda client service object as show. Replace *IDENTITY_POOL_ID* with the `IdentityPoolId` of the Amazon Cognito identity pool you created in the Create the AWS resources (p. 342) topic of this tutorial.

In the `LambdaApp` folder, create a file name `index.js`, and paste the content below into it.

```
// Load the required clients and packages.
const { InvokeCommand } = require ("@aws-sdk/client-lambda" );
const { lambdaClient } = require ( "../libs/lambdaClient" );

// Set the parmaeters.
const params={
  // The name of the AWS Lambda function.
  FunctionName: "LAMBDA_FUNCTION",
  InvocationType: "RequestResponse",
  LogType: "None"
}

// Call the Lambda function.
window.createTable = async () => {
  try {
    const data = await lambdaClient.send(new InvokeCommand(params));
    console.log("Table Created", data);
    document.getElementById('message').innerHTML = "Success, table created"
  } catch (err) {
    console.log("Error", err);
  }
};
```

In the parameters, replace *LAMBDA_FUNCTION* with a name unique in your AWS account, for example `createTable`.

This code example is available here on GitHub.

Finally, run the following at the command prompt to bundle the JavaScript for this example in a file named `main.js`.

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

> **Note**
> For information about installing webpack, see Bundling applications with webpack (p. 42).

Create the Lambda function

This topic is part of a tutorial that demonstrates how to create, deploy, and run a Lambda function using the AWS SDK for JavaScript. To start at the beginning of the tutorial, see Creating and using Lambda functions (p. 342).

In the root of your project, create an `mylambdafunction.js` file, and copy and paste the content below into it. Replace *REGION* with the AWS region.

Run the following command to bundle the Lambda function and the required Amazon Web Services modules.

```
webpack mylamdbafunction.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.

Compress the bundled output file, `index.js`, into a ZIP file named `my-lambda-function.zip`.

Upload `my-lambda-function.zip` to the Amazon S3 bucket you created in the Create the AWS resources (p. 342) topic of this tutorial.

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

// Set the parameters.
const params = {
    AttributeDefinitions: [
        {
            AttributeName: "Season", //ATTRIBUTE_NAME_1
            AttributeType: "N", //ATTRIBUTE_TYPE
        },
        {
            AttributeName: "Episode", //ATTRIBUTE_NAME_2
            AttributeType: "N", //ATTRIBUTE_TYPE
        },
    ],
    KeySchema: [
        {
            AttributeName: "Season", //ATTRIBUTE_NAME_1
            KeyType: "HASH",
        },
        {
            AttributeName: "Episode", //ATTRIBUTE_NAME_2
            KeyType: "RANGE",
        },
    ],
    ProvisionedThroughput: {
        ReadCapacityUnits: 1,
        WriteCapacityUnits: 1,
    },
    TableName: "TABLE_NAME", //TABLE_NAME
    StreamSpecification: {
        StreamEnabled: false,
    },
};

exports.handler = async(event, context, callback) => {
    try {
        const data = await dynamoClient.send(new CreateTableCommand(params));
        console.log("Table Created", data);
    } catch (err) {
        console.log("Error", err);
    }
};
```

This code example is available here on GitHub.

Deploy the Lambda function

This topic is part of a tutorial that demonstrates how to create, deploy, and run a Lambda function using the AWS SDK for JavaScript. To start at the beginning of the tutorial, see Creating and using Lambda functions (p. 342).

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 *KEY* with the name of name 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 (p. 342) topic of this tutorial. Replace *LAMBDA_FUNCTION* with the same name you gave the function in the `/Lambda/index.js` in the Prepare the browser script (p. 345) topic of this tutorial.

```
// Load the Lambda client.
const {
    LambdaClient,
    CreateFunctionCommand
} = require("@aws-sdk/client-lambda");

//Set the AWS Region.
const REGION = "REGION"; //e.g. "us-east-1"

// Instantiate an AWS 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: "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: "Creates an Amazon DynamoDB table.",
};

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.

To run the app, open the `index.html` in the Amazon S3 bucket that hosts the application. To do this, go open the Amazon S3 bucket in the console, select the bucket, and choose the **Object URL**.

Delete the resources

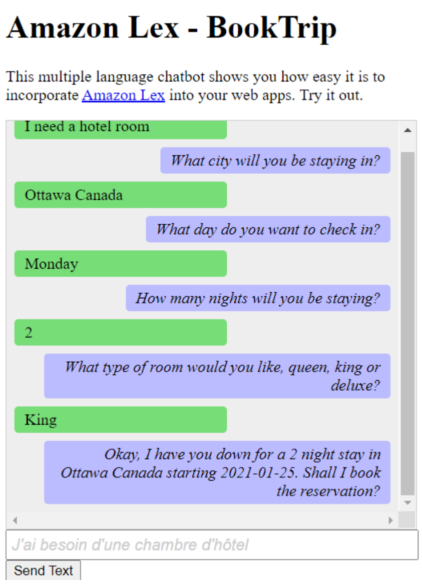
This topic is part of a tutorial that demonstrates how to create, deploy, and run a Lambda function using the AWS SDK for JavaScript. To start at the beginning of the tutorial, see Creating and using Lambda functions (p. 342).

When you finish the tutorial, you should delete the resources so you do not incur any unnecessary charges. You can do this by deleting the AWS CloudFormation stack you created in the Create the AWS resources (p. 342) topic of this tutorial.

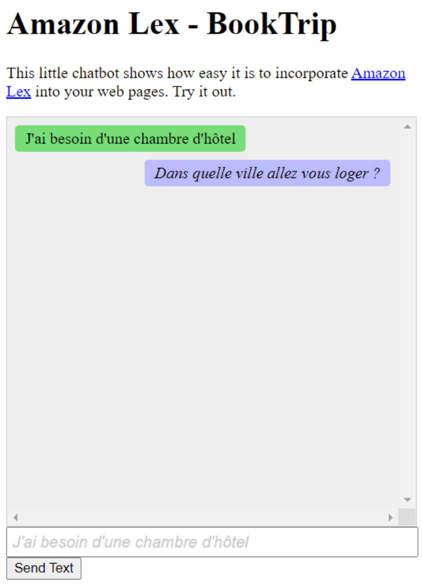
Because you created the DynamoDB table, you must delete it manually. For more information, see tsee the `ddb_deletetable.js` code sample here on GitHub. Then you can delete the remaining resources using either the Amazon Web Services Management Console or the AWS CLI. Instructions on how to modify the stack, or to delete the stack and its associated resources when you have finished the tutorial, see here on GitHub.

Building an Amazon Lex chatbot

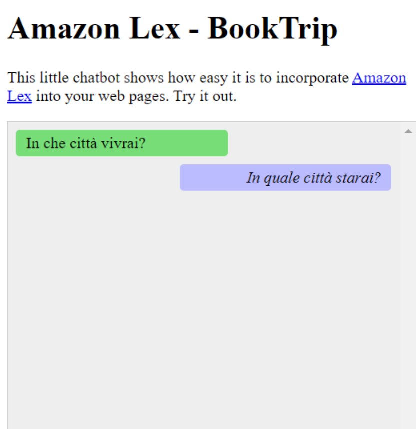
You can create an Amazon Lex chatbot within a web application to engage your web site visitors. An Amazon Lex chatbot is functionality that performs on-line chat conversation with users without providing direct contact with a person. For example, the following illustration shows an Amazon Lex chatbot that engages a user about booking a hotel room.



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.



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.

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:

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 (p. 349).

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 credentials JSON file, see Loading credentials in Node.js from the shared credentials file (p. 33).

> **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 sytax, please refer to JavaScript ES6/CommonJS syntax (p. 56)

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 (p. 349).

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 developer 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 (CDK) Developer Guide.

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

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

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

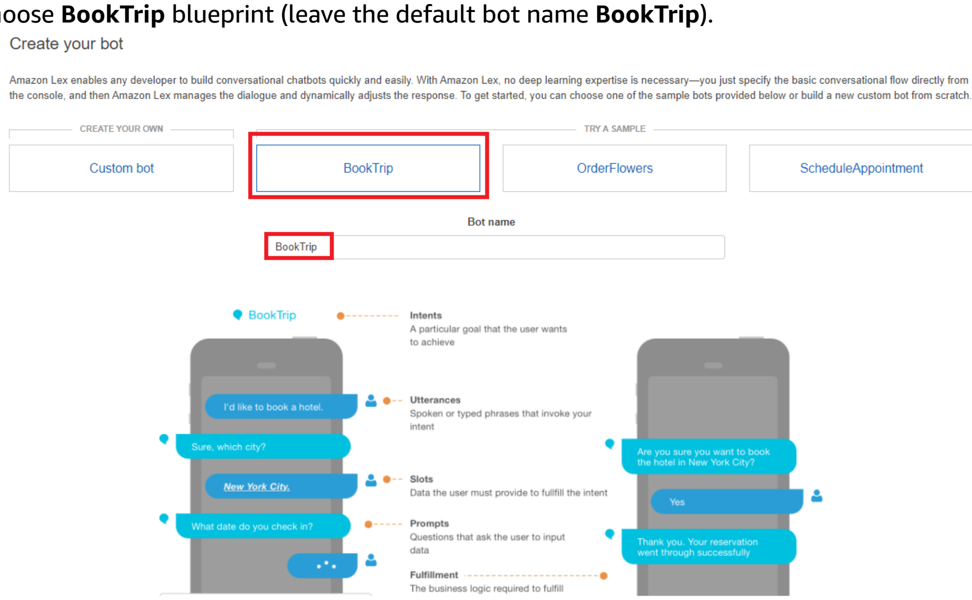
   For more information on the `create-stack` command parameters, see the AWS CLI Command Reference guide, and the AWS CloudFormation User Guide.

   To view the resources created, open the Amazon Lex console, choose the stack, and select the **Resources** tab.

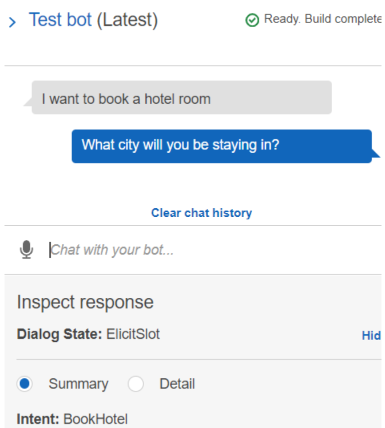
Create an Amazon Lex bot

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*.



- 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 (p. 349).

Create a file named `index.html`. Copy and paste the code below in to `index.html`. This HTML references `main.js`. This is a bundled version of index.js, which includes the required AWS SDK for JavaScript modules. You'll create this file in Create the HTML (p. 353). `index.html` also references `style.css`, which adds the styles.

```
<!DOCTYPE html>
<head>
    <title>Amazon Lex - Sample Application (BookTrip)</title>
    <link type="text/css" rel="stylesheet" href="style.css">
</head>

<body>
<h1 id="title">Amazon Lex - BookTrip</h1>
<p id="intro">
    This multiple language chatbot shows you how easy it is to incorporate
    <a href="https://aws.amazon.com/lex/" title="Amazon Lex (product)" target="_new">Amazon
 Lex</a> into your web apps.  Try it out.
</p>
<div id="conversation"></div>
<input type="text" id="wisdom" size="80" value="" placeholder="J'ai besoin d'une chambre
 d'hôtel">
<br>
<button onclick="createResponse()">Send Text</button>
<script type="text/javascript" src="./main.js"></script>
</body>
```

This code is also available here on GitHub.

Create the browser script

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 (p. 349).

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 (p. 351). 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 `transcribeClient.js`. Paste the approriate code below into each, and replace *REGION* and *IDENTITY_POOL_ID* in each.

Note

Use the ID of the Amazon Cognito identity pool you created in Create the AWS resources using AWS CloudFormation (p. 351).

```
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"; //e.g. "us-east-1"
const IdentityPoolId = "IDENTITY_POOL_ID";
const comprehendClient = new ComprehendClient({
  region: REGION,
  credentials: fromCognitoIdentityPool({
    client: new CognitoIdentityClient({ region: REGION }),
    identityPoolId: IdentityPoolId
  }),
});
```

```
import { CognitoIdentityClient } from "@aws-sdk/client-cognito-identity";
import { fromCognitoIdentityPool } from "@aws-sdk/credential-provider-cognito-identity";
import { LexClient } from "@aws-sdk/client-lex";
const REGION = "REGION"; //e.g. "us-east-1"
const IdentityPoolId = "IDENTITY_POOL_ID";
const lexClient = new LexRuntimeServiceClient({
  region: REGION,
  credentials: fromCognitoIdentityPool({
    client: new CognitoIdentityClient({ region: REGION }),
    identityPoolId: IdentityPoolId
  }),
});
```

```
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"; //e.g. "us-east-1"
const IdentityPoolId = "IDENTITY_POOL_ID";
const translateClient = new TranslateClient({
  region: REGION,
  credentials: fromCognitoIdentityPool({
    client: new CognitoIdentityClient({ region: REGION }),
    identityPoolId: IdentityPoolId
  }),
});
```

This code is available here on GitHub..

Next, create an `index.html` 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 {comprehendClient} from ".libs/comprehendClient.js";
import {translateClient} from ".libs/translateClient.js";

var g_text = "";
// set the focus to the input box
document.getElementById("wisdom").focus();

function showRequest(daText) {
  var conversationDiv = document.getElementById("conversation");
  var requestPara = document.createElement("P");
  requestPara.className = "userRequest";
  requestPara.appendChild(document.createTextNode(g_text));
  conversationDiv.appendChild(requestPara);
  conversationDiv.scrollTop = conversationDiv.scrollHeight;
};

function showResponse(lexResponse) {
  var conversationDiv = document.getElementById("conversation");
  var responsePara = document.createElement("P");
  responsePara.className = "lexResponse";

  var lexTextResponse = lexResponse;

  responsePara.appendChild(document.createTextNode(lexTextResponse));
  responsePara.appendChild(document.createElement("br"));
  conversationDiv.appendChild(responsePara);
  conversationDiv.scrollTop = conversationDiv.scrollHeight;
};

function handletext(text) {
  g_text = text;
  var xhr = new XMLHttpRequest();
  xhr.addEventListener("load", loadNewItems, false);
  xhr.open("POST", "../text", true); // A Spring MVC controller
  xhr.setRequestHeader("Content-type", "application/x-www-form-urlencoded"); //necessary
  xhr.send("text=" + text);
};

function loadNewItems(event) {
```

```
  var msg = event.target.responseText;
  showRequest();
  showResponse(msg);

  // 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;

    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 = {
          botAlias: "BOT_ALIAS",
          botName: "BOT_NAME",
          inputText: data.TranslatedText,
          userId: "USER_ID" // For example, 'chatbot-demo'.
        };
        try {
          const data = await lexClient.send(new PostTextCommand(lexParams));
          console.log("Success. Response is: ", data.message);
          document.getElementById("conversation").innerHTML = data.message;
        } 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);
    }
  }
};
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 (p. 351) for this example to install webpack.

   Note
   For information about*webpack*, see Bundling applications with webpack (p. 42).
2. Run the the following in the command line to bundle the JavaScript for this example into a file called `<index.js>` :

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

Next steps

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 (p. 349).

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 (p. 351) topic of this tutorial,as follows:

1. Open the AWS CloudFormation console.
2. On the **Stacks** page, select the stack.
3. Choose **Delete**.

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

Creating an example messaging application

You can create an AWS application that sends and retrieves messages by using the AWS SDK for JavaScript and Amazon Simple Queue Service (Amazon SQS). Messages are stored in a first in, first out (FIFO) queue that ensures that the order of the messages is consistent. For example, the first message that's stored in the queue is the first message read from the queue.

Note
For more information about Amazon SQS, see What is Amazon Simple Queue Service?

In this tutorial, you create a Node.js application named AWS Messaging.

Cost to complete: The AWS services included in this document are included in the AWS Free Tier.

Note: Be sure to terminate all of the resources you create while going through this tutorial to ensure that you're not charged.

To build the app:

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 (p. 357).

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 credentials JSON file, see Loading credentials in Node.js from the shared credentials file (p. 33).

> **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 sytax, please refer to JavaScript ES6/CommonJS syntax (p. 56)

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 (p. 357).

This tutorial requires the following resources.

- An unautenticated 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 Developer 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 developer 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 (CDK) Developer Guide.

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

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

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

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

   To view the resources created, open AWS CloudFormation in the AWS management console, choose the stack, and select the **Resources** tab.

Understand the AWS Messaging application

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 (p. 357).

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 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 (p. 357).

Now you create the HTML files that are required for the application's graphical user interface (GUI). Create a file named `index.html`. Copy and paste the code below in to `index.html`. This HTML references `main.js`. This is a bundled version of index.js, which includes the required AWS SDK for JavaScript modules.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org" xmlns:sec="http://www.thymeleaf.org/thymeleaf-
extras-springsecurity3">
```

```
<head>
    <meta charset="utf-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <link rel="icon" href="./images/favicon.ico" />
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/
bootstrap.min.css"/>
    <link rel="stylesheet" href="./css/styles.css"/>
    <script src="https://code.jquery.com/jquery-1.12.4.min.js"></script>
    <script src="https://code.jquery.com/ui/1.11.4/jquery-ui.min.js"></script>
    <script src="./js/main.js"></script>
    <style>
        .messageelement {
            margin: auto;
            border: 2px solid #dedede;
            background-color: #D7D1D0 ;
            border-radius: 5px;
            max-width: 800px;
            padding: 10px;
            margin: 10px 0;
        }

        .messageelement::after {
            content: "";
            clear: both;
            display: table;
        }

        .messageelement img {
            float: left;
            max-width: 60px;
            width: 100%;
            margin-right: 20px;
            border-radius: 50%;
        }

        .messageelement img.right {
            float: right;
            margin-left: 20px;
            margin-right:0;
        }
    </style>
</head>
<body>

<div class="container">
    <h2>AWS Sample Messaging Application</h2>
    <div id="messages">

    </div>

    <div class="input-group mb-3">
        <div class="input-group-prepend">
            <span class="input-group-text" id="basic-addon1">Sender:</span>
        </div>
        <select name="cars" id="username">
            <option value="Scott">Brian</option>
            <option value="Tricia">Tricia</option>
        </select>
    </div>

    <div class="input-group">
        <div class="input-group-prepend">
            <span class="input-group-text">Message:</span>
        </div>
        <textarea class="form-control" id="textarea" aria-label="With textarea"></textarea>
```

```
        <button type="button" onclick="pushMessage()" id="send" class="btn btn-
success">Send</button>
        <button type="button" onclick="purge()" id="refresh" class="btn btn-
success">Purge</button>
    </div>
    <!-- All of these child items are hidden and only displayed in a FancyBox
 ------------------------------------------------------->
    <div id="hide" style="display: none">

        <div id="base" class="messageelement">
            <img src="../public/images/av2.png"  alt="Avatar" class="right"
 style="width:100%;">
            <p id="text">Excellent! So, what do you want to do today?</p>
            <span class="time-right">11:02</span>
        </div>

    </div>
</div>
</body>
</html>
```

This code is also available here on GitHub.

Creating the browser script

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 (p. 357).

In this topic, you create a the browser script for the app. When you have created the browser script, you bundle it into a file called `main.js` as described in Bundling the JavaScript (p. 365).

Create a file named `index.js`. Copy and paste the code from here on GitHub into it.

This code is explained in the following sections:

Configuration

First, create a `libs` directory create the required Amazon SQS client object by creating a files named `sqsClient.js`. Replace ***REGION*** and ***IDENTITY_POOL_ID*** in each.

> **Note**
> Use the ID of the Amazon Cognito identity pool you created in Create the AWS resources (p. 358).

```
import { CognitoIdentityClient } from "@aws-sdk/client-cognito-identity";
import { fromCognitoIdentityPool } from "@aws-sdk/credential-provider-cognito-identity";
import {SQSClient} from "@aws-sdk/client-sqs"'
const REGION = "REGION"; //e.g. "us-east-1"
const IdentityPoolId = "IDENTITY_POOL_ID";
const sqsClient = new SQSClient({
  region: REGION,
  credentials: fromCognitoIdentityPool({
    client: new CognitoIdentityClient({ region: REGION }),
```

```
    identityPoolId: IdentityPoolId
  }),
});
```

In the `index.js`, import the required AWS SDK for JavaScript modules and commands. Replace *SQS_QUEUE_NAME* with the name of the Amazon SQS Queue you created in the Create the AWS resources (p. 358).

```
import {
  GetQueueUrlCommand,
  SendMessageCommand,
  ReceiveMessageCommand,
  PurgeQueueCommand,
} from "@aws-sdk/client-sqs";
import { sqsClient } from "./libs/sqsClient.js";

const QueueName = "SQS_QUEUE_NAME"; // The Amazon SQS queue name, which must end in .fifo
 for this example.
```

populateChat

The `populateChat` function onload automatically retrieves the URL for the Amazon SQS Queue, and retrieves all messages in the queue, and displays them.

```
$(function () {
  populateChat();
});

const populateChat = async () => {
  try {
    // Set the Amazon SQS Queue parameters.
    const queueParams = {
      QueueName: QueueName,
      Attributes: {
        DelaySeconds: "60",
        MessageRetentionPeriod: "86400",
      },
    };
    // Get the Amazon SQS Queue URL.
    const data = await sqsClient.send(new GetQueueUrlCommand(queueParams));
    console.log("Success. The URL of the SQS Queue is: ", data.QueueUrl);
    // Set the parameters for retrieving the messages in the Amazon SQS Queue.
    var getMessageParams = {
      QueueUrl: data.QueueUrl,
      MaxNumberOfMessages: 10,
      MessageAttributeNames: ["All"],
      VisibilityTimeout: 20,
      WaitTimeSeconds: 20,
    };
    try {
      // Retrieve the messages from the Amazon SQS Queue.
      const data = await sqsClient.send(
        new ReceiveMessageCommand(getMessageParams)
      );
      console.log("Successfully retrieved messages", data.Messages);

      // Loop through messages for user and message body.
      var i;
      for (i = 0; i < data.Messages.length; i++) {
        const name = data.Messages[i].MessageAttributes.Name.StringValue;
```

```
        const body = data.Messages[i].Body;
        // Create the HTML for the message.
        var userText = body + "<br><br><b>" + name;
        var myTextNode = $("#base").clone();
        myTextNode.text(userText);
        var image_url;
        var n = name.localeCompare("Scott");
        if (n == 0) image_url = "./images/av1.png";
        else image_url = "./images/av2.png";
        var images_div =
          '<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);
    }
  } catch (err) {
    console.log("Error retrieving SQS queue URL: ", err);
  }
};
```

Push messages

The user selects their name and enters their message, and submits the message, which initiates the `pushMessage` function. `pushMessage` retrieves the Amazon SQS Queue Url, and then sends a message with a unique message ID value (a GUID) the message text, and the user to the Amazon SQS Queue. It then retrieves all the messages from the Amazon SQS Queue and displays them.

```
const pushMessage = async () => {
  // Get and convert user and message input.
  var user = $("#username").val();
  var message = $("#textarea").val();

  // Create random deduplication ID.
  var dt = new Date().getTime();
  var uuid = "xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx".replace(/[xy]/g, function (
    c
  ) {
    var r = (dt + Math.random() * 16) % 16 | 0;
    dt = Math.floor(dt / 16);
    return (c == "x" ? r : (r & 0x3) | 0x8).toString(16);
  });

  try {
    // Set the Amazon SQS Queue parameters.
    const queueParams = {
      QueueName: QueueName,
      Attributes: {
        DelaySeconds: "60",
        MessageRetentionPeriod: "86400",
      },
    };
    const data = await sqsClient.send(new GetQueueUrlCommand(queueParams));
    console.log("Success. The URL of the SQS Queue is: ", data.QueueUrl);
    // Set the parameters for the message.
    var messageParams = {
      MessageAttributes: {
```

```
        Name: {
          DataType: "String",
          StringValue: user,
        },
      },
      MessageBody: message,
      MessageDeduplicationId: uuid,
      MessageGroupId: "GroupA",
      QueueUrl: data.QueueUrl,
    };
    const result = await sqsClient.send(new SendMessageCommand(messageParams));
    console.log("Success", result.MessageId);

    // Set the parameters for retrieving all messages in the SQS queue.
    var getMessageParams = {
      QueueUrl: data.QueueUrl,
      MaxNumberOfMessages: 10,
      MessageAttributeNames: ["All"],
      VisibilityTimeout: 20,
      WaitTimeSeconds: 20,
    };

    // Retrieve messages from SQS Queue.
    const final = await sqsClient.send(
      new ReceiveMessageCommand(getMessageParams)
    );
    console.log("Successfully retrieved", final.Messages);
    $("#messages").empty();
    // Loop through messages for user and message body.
    var i;
    for (i = 0; i < final.Messages.length; i++) {
      const name = final.Messages[i].MessageAttributes.Name.StringValue;
      const body = final.Messages[i].Body;
      // Create the HTML for the message.
      var userText = body + "<br><br><b>" + name;
      var myTextNode = $("#base").clone();
      myTextNode.text(userText);
      var image_url;
      var n = name.localeCompare("Scott");
      if (n == 0) image_url = "./images/av1.png";
      else image_url = "./images/av2.png";
      var images_div =
        '<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 comlete 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 (p. 358) for this example to install webpack.

   Note
   For information about*webpack*, see Bundling applications with webpack (p. 42).
2. Run the the following in the command line to bundle the JavaScript for this example into a file called `<index.js>` :

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

Next steps

Congratulations! You have created and deployed the AWS Messaging application that uses Amazon SQS. As stated at the beginning of this tutorial, be sure to terminate all of the resources you create while going through this tutorial to ensure that you're no longer charged for them. You can do this by deleting the AWS CloudFormation stack you created in the Create the AWS resources (p. 358) 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.

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

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. This content includes 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 user accounts with 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 recommend TLS 1.2 or later.
- 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 personal 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 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 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 for this AWS Product or Service

AWS Identity and Access Management (IAM) is an Amazon Web Services (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 resources in AWS services. IAM is an AWS service that you can use with no additional charge.

To use this AWS product or service to access AWS, you need an AWS account and AWS credentials. To increase the security of your AWS account, we recommend that you use an *IAM user* to provide access credentials instead of using your AWS account credentials.

For details about working with IAM, see AWS Identity and Access Management.

For an overview of IAM users and why they are important for the security of your account, see AWS Security Credentials in the Amazon Web Services General Reference.

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.

Compliance Validation for this AWS Product or Service

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.

The security and compliance of AWS services is assessed by third-party auditors as part of multiple AWS compliance programs. These include SOC, PCI, FedRAMP, HIPAA, and others. AWS provides a frequently updated list of AWS services in scope of specific compliance programs at AWS Services in Scope by Compliance Program.

Third-party audit reports are available for you to download using AWS Artifact. For more information, see Downloading Reports in AWS Artifact.

For more information about AWS compliance programs, see AWS Compliance Programs.

Your compliance responsibility when using this AWS product or service to access an AWS service is determined by the sensitivity of your data, your organization's compliance objectives, and applicable laws and regulations. If your use of an AWS service is subject to compliance with standards such as HIPAA, PCI, or FedRAMP, AWS provides resources to help:

- Security and Compliance Quick Start Guides – Deployment guides that discuss architectural considerations and provide steps for deploying security-focused and compliance-focused baseline environments on AWS.
- Architecting for HIPAA Security and Compliance Whitepaper – A whitepaper that describes how companies can use AWS to create HIPAA-compliant applications.
- AWS Compliance Resources – A collection of workbooks and guides that might apply to your industry and location.
- AWS Config – A service that assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.
- AWS Security Hub – A comprehensive view of your security state within AWS that helps you check your compliance with security industry standards and best practices.

Resilience for this AWS Product or Service

The Amazon Web Services (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 Availability Zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information about AWS Regions and Availability Zones, see AWS Global Infrastructure.

This AWS product or service follows the shared responsibility model through the specific Amazon Web Services (AWS) services it supports. For AWS service security information, see the AWS service security documentation page and AWS services that are in scope of AWS compliance efforts by compliance program.

Infrastructure Security for this AWS Product or Service

This AWS product or service 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.

Enforcing TLS 1.2

To add increased security when communicating with AWS services, configure the AWS SDK for JavaScript to use TLS 1.2 or later.

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 8.0.0 and later use a minimum version of OpenSSL 1.0.2, which supports TLS 1.2. The SDK for JavaScript defaults to use TLS 1.2 when available.

Verify the version of OpenSSL and TLS

To get the version of OpenSSL used by Node.js on your computer, run the following command.

```
node -p process.versions
```

The version of OpenSSL in the list is the version used by Node.js, as shown in the following example.

```
openssl: '1.1.1d'
```

To get the version of TLS used by Node.js on your computer, start the Node shell and run the following commands, in order.

```
> var tls = require("tls");
> var tlsSocket = new tls.TLSSocket();
> tlsSocket.getProtocol();
```

The last command outputs the TLS version, as shown in the following example.

```
'TLSv1.3'
```

Node.js defaults to use this version of TLS, and tries to negotiate another version of TLS if a call is not successful.

Enforce a minimum version of TLS

Node.js negotiates a version of TLS when a call fails. You can enforce the minimum allowable TLS version during this negotiation, either when running a script from the command line or per request in your JavaScript code.

To specify the minimum TLS version from the command line, you must use Node.js version 11.0.0 or later. To install a specific Node.js version, first install Node Version Manager (nvm) using the steps found at Node version manager installing and updating. Then run the following commands to install and use a specific version of Node.js.

```
nvm install 11
nvm use 11
```

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.

```
const https = require("https");
const {NodeHttpHandler} = require("@aws-sdk/node-http-handler");
const {DynamoDBClient} = require("@aws-sdk/client-dynamodb");

const client = new DynamoDBClient({
    region: "us-west-2",
    requestHandler: new NodeHttpHandler({
        httpsAgent: new https.Agent(
            {
                secureProtocol: 'TLSv1_2_method'
            }
        )
    })
});
```

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.

Document history for AWS SDK for JavaScript version 3

Document History

- **Latest documentation update:**November 09, 2020

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.

update-history-change	update-history-description	update-history-date
Updated AWS Lambda tutorial (p. 298)	Added tutorial demonstrating how to build a browser-based application for submitting data to a Amazon DynamoDB table.	October 20, 2020
Setting credentials in Node.js topic updated (p. 28)	Update topic about setting credentials in Node.js for AWS SDK for JavaScript V3.	October 20, 2020
Migrating to V3 (p. 24)	Added topic to describe how to migrate to AWS SDK for JavaScript V3.	October 20, 2020
Getting Started (p. 8)	Updated topics for getting started in the browser and getting started with Node.js for AWS SDK for JavaScript V3.	October 20, 2020
Browser builder (p. 371)	Information about AWS Broswer Builder was removed because it is not required for AWS SDK for JavaScript V3.	October 20, 2020
Amazon Transcribe service examples updated (p. 284)	Updated Amazon Transcribe service examples for AWS SDK for JavaScript V3.	October 20, 2020
Amazon Simple Storage Service service examples updated (p. 173)	Updated Amazon Simple Storage Service service examples for AWS SDK for JavaScript V3.	October 20, 2020
Amazon Simple Queue Service service examples updated (p. 268)	Updated Amazon Simple Queue Service service examples for AWS SDK for JavaScript V3.	October 20, 2020
Amazon Simple Notification Service service examples updated (p. 248)	Updated Amazon Simple Notification Service service examples for AWS SDK for JavaScript V3.	October 20, 2020

Amazon Simple Email Service service examples updated (p. 223)	Updated Amazon Simple Email Service service examples for AWS SDK for JavaScript V3.	October 20, 2020
Amazon S3 Glacier service examples updated (p. 137)	Updated Amazon S3 Glacier service examples for AWS SDK for JavaScript V3.	October 20, 2020
Amazon Redshift service examples updated (p. 291)	Updated Amazon Redshift service examples for AWS SDK for JavaScript V3.	October 20, 2020
Amazon Lex service examples updated (p. 170)	Updated Amazon Lex service examples for AWS SDK for JavaScript V3.	October 20, 2020
Amazon Kinesis service examples updated (p. 162)	Updated Amazon Kinesis service examples for AWS SDK for JavaScript V3.	October 20, 2020
Amazon Elastic Compute Cloud service examples updated (p. 103)	Updated Amazon Elastic Compute Cloud service examples for AWS SDK for JavaScript V3.	October 20, 2020
Amazon DynamoDB service examples updated (p. 78)	Updated Amazon DynamoDB service examples for AWS SDK for JavaScript V3.	October 20, 2020
Amazon CloudWatch service examples updated (p. 58)	Updated Amazon CloudWatch service examples for AWS SDK for JavaScript V3.	October 20, 2020
AWS Elemental MediaConvert service examples updated (p. 122)	Updated AWS Elemental MediaConvert service examples for AWS SDK for JavaScript V3.	October 20, 2020
AWS Identity and Access Management service examples updated (p. 140)	Updated AWS Identity and Access Management service examples for AWS SDK for JavaScript V3.	October 20, 2020
AWS Lambda service examples updated (p. 170)	Updated AWS Lambda service examples for AWS SDK for JavaScript V3.	October 20, 2020
AWS SDK for JavaScript V3 Developer Guide preview (p. 371)	Released pre-release version of the AWS SDK for JavaScript V3 Developer Guide.	October 19, 2020