



Developer Guide for SDK v2

AWS SDK for JavaScript



AWS SDK for JavaScript: Developer Guide for SDK v2

Copyright © 2025 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

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

Table of Contents

.....	ix
What Is the AWS SDK for JavaScript?	1
Maintenance and support for SDK major versions	1
Using the SDK with Node.js	2
Using the SDK with AWS Amplify	2
Using the SDK with Web Browsers	2
Common Use Cases	3
About the Examples	3
Getting Started	4
Getting Started in a Browser Script	4
The Scenario	4
Step 1: Create an Amazon Cognito Identity Pool	5
Step 2: Add a Policy to the Created IAM Role	6
Step 3: Create the HTML Page	7
Step 4: Write the Browser Script	8
Step 5: Run the Sample	9
Full Sample	10
Possible Enhancements	11
Getting Started in Node.js	12
The Scenario	12
Prerequisite Tasks	12
Step 1: Install the SDK and Dependencies	13
Step 2: Configure Your Credentials	13
Step 3: Create the Package JSON for the Project	14
Step 4: Write the Node.js Code	15
Step 5: Run the Sample	16
Setting Up the SDK for JavaScript	17
Prerequisites	17
Setting Up an AWS Node.js Environment	17
Web Browsers Supported	18
Installing the SDK	19
Installing Using Bower	20
Loading the SDK	20
Upgrading From Version 1	21

Automatic Conversion of Base64 and Timestamp Types on Input/Output	21
Moved response.data.RequestId to response.requestId	22
Exposed Wrapper Elements	23
Dropped Client Properties	28
Configuring the SDK for JavaScript	29
Using the Global Configuration Object	29
Setting Global Configuration	30
Setting Configuration Per Service	32
Immutable Configuration Data	32
Setting the AWS Region	32
In a Client Class Constructor	33
Using the Global Configuration Object	33
Using an Environment Variable	33
Using a Shared Config File	33
Order of Precedence for Setting the Region	34
Specifying Custom Endpoints	34
Endpoint String Format	34
Endpoints for the ap-northeast-3 Region	35
Endpoints for MediaConvert	35
SDK authentication with AWS	35
Start an AWS access portal session	37
More authentication information	37
Setting Credentials	38
Best Practices for Credentials	38
Setting Credentials in Node.js	39
Setting Credentials in a Web Browser	44
Locking API Versions	54
Getting API Versions	54
Node.js Considerations	55
Using Built-In Node.js Modules	55
Using NPM Packages	55
Configuring maxSockets in Node.js	56
Reusing Connections with Keep-Alive in Node.js	57
Configuring Proxies for Node.js	58
Registering Certificate Bundles in Node.js	59
Browser Script Considerations	59

Building the SDK for Browsers	60
Cross-Origin Resource Sharing (CORS)	63
Bundling with Webpack	67
Installing Webpack	67
Configuring Webpack	68
Running Webpack	69
Using the Webpack Bundle	70
Importing Individual Services	70
Bundling for Node.js	71
Working with Services	73
Creating and Calling Service Objects	74
Requiring Individual Services	75
Creating Service Objects	76
Locking the API Version of a Service Object	77
Specifying Service Object Parameters	77
Logging AWS SDK for JavaScript Calls	78
Using a Third-Party Logger	78
Calling Services Asynchronously	79
Managing Asynchronous Calls	79
Using a Callback Function	80
Using a Request Object Event Listener	82
Using async/await	87
Using Promises	88
Using the Response Object	90
Accessing Data Returned in the Response Object	91
Paging Through Returned Data	91
Accessing Error Information from a Response Object	92
Accessing the Originating Request Object	92
Working with JSON	93
JSON as Service Object Parameters	94
Returning Data as JSON	94
Retries	95
Exponential backoff based retry behavior	95
SDK for JavaScript Code Examples	99
Amazon CloudWatch Examples	99
Creating Alarms in Amazon CloudWatch	100

Using Alarm Actions in Amazon CloudWatch	104
Getting Metrics from Amazon CloudWatch	108
Sending Events to Amazon CloudWatch Events	111
Using Subscription Filters in Amazon CloudWatch Logs	116
Amazon DynamoDB Examples	121
Creating and Using Tables in DynamoDB	122
Reading and Writing A Single Item in DynamoDB	127
Reading and Writing Items in Batch in DynamoDB	130
Querying and Scanning a DynamoDB Table	134
Using the DynamoDB Document Client	137
Amazon EC2 Examples	143
Creating an Amazon EC2 Instance	144
Managing Amazon EC2 Instances	147
Working with Amazon EC2 Key Pairs	153
Using Regions and Availability Zones with Amazon EC2	156
Working with Security Groups in Amazon EC2	158
Using Elastic IP Addresses in Amazon EC2	163
MediaConvert Examples	167
Getting Your Region-Specific Endpoint	167
Creating and Managing Jobs	169
Using Job Templates	177
AWS IAM Examples	185
Managing IAM Users	186
Working with IAM Policies	191
Managing IAM Access Keys	197
Working with IAM Server Certificates	202
Managing IAM Account Aliases	206
Amazon Kinesis Example	209
Capturing Web Page Scroll Progress with Amazon Kinesis	210
Amazon S3 Examples	217
Amazon S3 Browser Examples	218
Amazon S3 Node.js Examples	246
Amazon SES Examples	267
Managing Identities	268
Working with Email Templates	274
Sending Email Using Amazon SES	280

Using IP Address Filters	286
Using Receipt Rules	290
Amazon SNS Examples	295
Managing Topics	296
Publishing Messages to a Topic	302
Managing Subscriptions	304
Sending SMS Messages	310
Amazon SQS Examples	316
Using Queues in Amazon SQS	317
Sending and Receiving Messages in Amazon SQS	321
Managing Visibility Timeout in Amazon SQS	325
Enabling Long Polling in Amazon SQS	327
Using Dead Letter Queues in Amazon SQS	331
Tutorials	334
Tutorial: Setting Up Node.js on an Amazon EC2 Instance	334
Prerequisites	334
Procedure	334
Creating an Amazon Machine Image	336
Related Resources	336
API Reference and Changelog	337
SDK Changelog on GitHub	337
Migrate to v3	338
Security	339
Data protection	339
Identity and Access Management	340
Audience	341
Authenticating with identities	341
Managing access using policies	345
How AWS services work with IAM	347
Troubleshooting AWS identity and access	347
Compliance Validation	349
Resilience	350
Infrastructure Security	351
Enforcing a minimum version of TLS	351
Verify and enforce TLS in Node.js	352
Verify and enforce TLS in a browser script	355

Additional Resources **357**

AWS SDKs and Tools Reference Guide 357

JavaScript SDK Forum 357

JavaScript SDK and Developer Guide on GitHub 357

JavaScript SDK on Gitter 357

Document History **358**

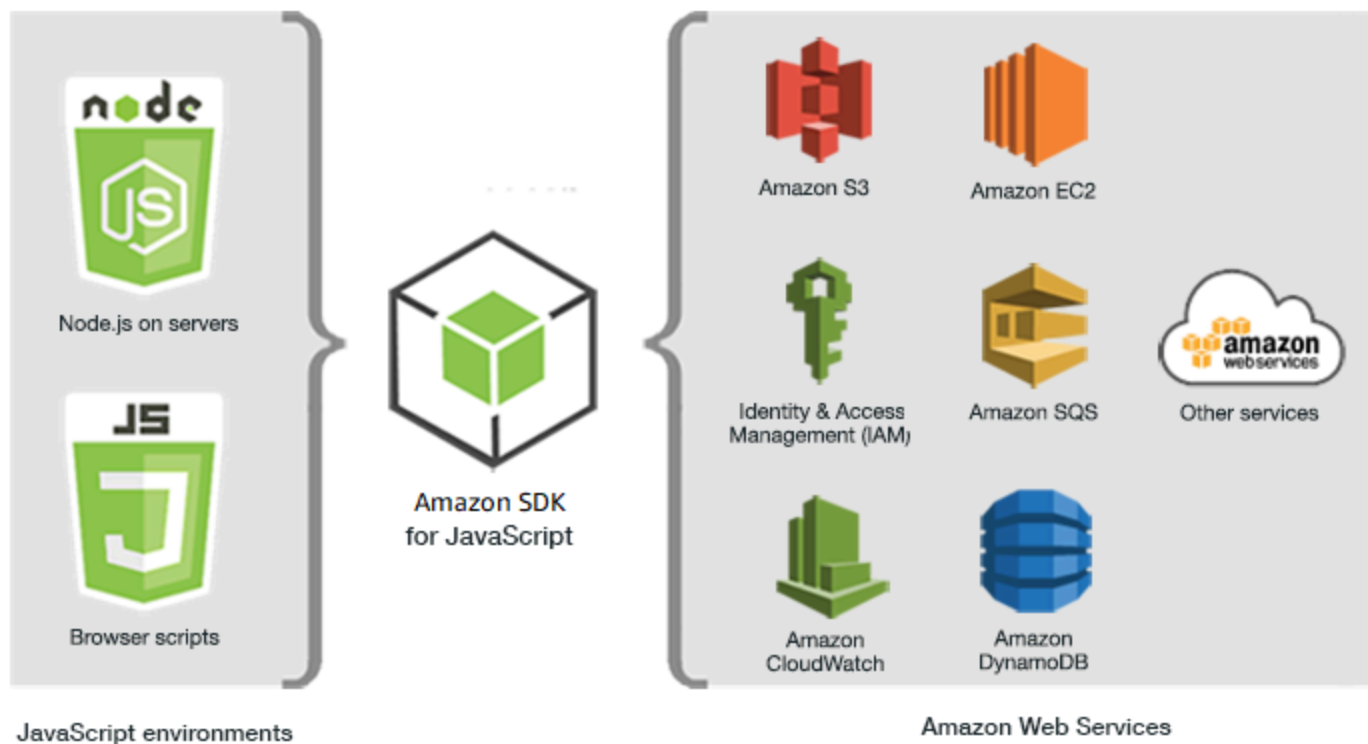
Document History 358

Earlier Updates 359

We [announced](#) the upcoming end-of-support for AWS SDK for JavaScript v2. We recommend that you migrate to [AWS SDK for JavaScript v3](#). For dates, additional details, and information on how to migrate, please refer to the linked announcement.

What Is the AWS SDK for JavaScript?

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



Not all services are immediately available in the SDK. To find out which services are currently supported by the AWS SDK for JavaScript, see <https://github.com/aws/aws-sdk-js/blob/master/SERVICES.md>. For information about the SDK for JavaScript on GitHub, see [Additional Resources](#).

Maintenance and support for SDK major versions

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

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

Using the SDK with Node.js

Node.js is a cross-platform runtime for running server-side JavaScript applications. You can set up Node.js on an Amazon 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, those differences will be called out.

Using the SDK with AWS Amplify

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

Note

Frameworks such as AWS Amplify might not offer the same browser support as the SDK for JavaScript. Check a 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*.

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, those differences will be called out.

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

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 Library](#).

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.

Topics

- [Getting Started in a Browser Script](#)
- [Getting Started in Node.js](#)

Getting Started in a Browser Script



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.

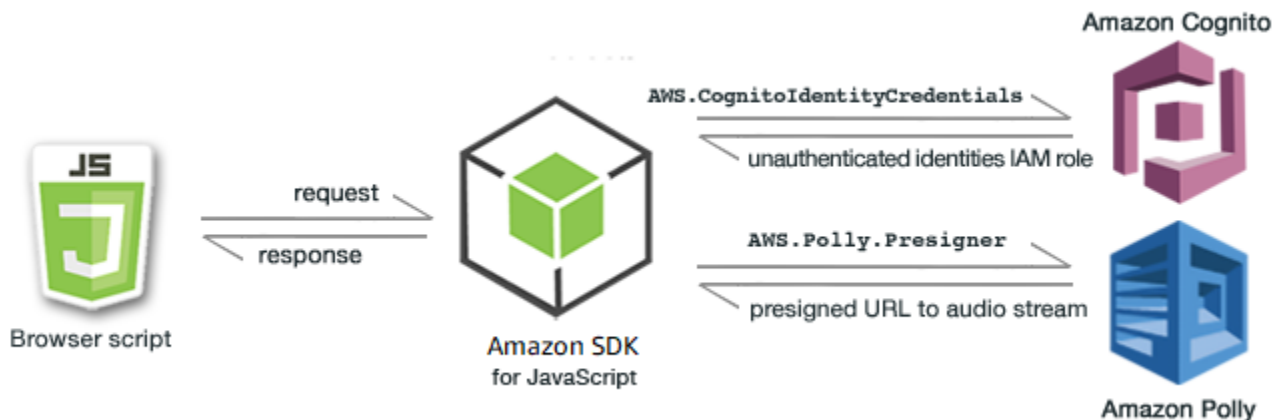
The Scenario

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](#).

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

Note

Playback of the synthesized speech in this example depends on running in a browser that supports HTML 5 audio.



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

- [AWS.CognitoIdentityCredentials](#) constructor
- [AWS.Polly.Presigner](#) 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 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. For more information about adding a Amazon Cognito identity pool, see [Tutorial: Creating an identity pool](#) in the *Amazon Cognito Developer Guide*.

To create an Amazon Cognito identity pool

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

5. In the **Add permissions** page for this role, find and then select the check box for **AmazonPollyReadOnly**.

 Note

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

6. Choose **Add permissions**.

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

Step 3: Create the HTML Page

The sample app consists of a single HTML page that contains the user interface and browser script. 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.)

For more information on the `<audio>` element, see [audio](#).

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

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


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.

Step 4: Write the Browser Script

The first thing to do when creating the browser script is to include the SDK for JavaScript by adding a `<script>` element after the `<audio>` element in the page. To find the current `SDK_VERSION_NUMBER`, see the API Reference for the SDK for JavaScript at [AWS SDK for JavaScript API Reference Guide](#).

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

Then add a new `<script type="text/javascript">` element after the SDK entry. You'll add the browser script to this element. Set the AWS Region and credentials for the SDK. Next, create a function named `speakText()` that will be invoked as an event handler by the button.

To synthesize speech with Amazon Polly, you must provide 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. Replace *IDENTITY_POOL_ID* and *REGION* in the following code with values noted in [Step 1: Create an Amazon Cognito Identity Pool](#).

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

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

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

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

Amazon 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.

Create a new `AWS.Polly` service object. Then create the `AWS.Polly.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 `AWS.Polly` service object that you created to the `AWS.Polly.Presigner` constructor.

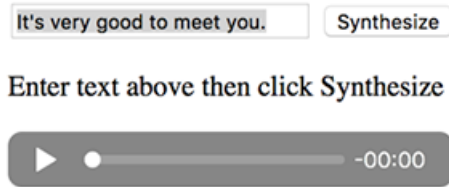
After you create the presigner object, call the `getSynthesizeSpeechUrl` method of that object, passing the speech parameters. If successful, this method returns the URL of the synthesized speech, which you then assign to the `<audio>` element for playback.

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

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

Step 5: Run the Sample

To run the sample app, load `polly.html` into a web browser. This is what the browser presentation should resemble.



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.

Full Sample

Here is the full HTML page with the browser script. It's also available [here on GitHub](#).

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

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

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

      // Function invoked by button click
      function speakText() {
```

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

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

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

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 create the package .json manifest for your project.
- How to install and include the modules that your project uses.
- How to create an Amazon Simple Storage Service (Amazon S3) service object from the AWS .S3 client class.
- How to create an Amazon S3 bucket and upload an object to that bucket.

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.

Because bucket names in Amazon S3 must be globally unique, this example includes a third-party Node.js module that generates a unique ID value that you can incorporate into the bucket name. This additional module is named `uuid`.

Prerequisite Tasks

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

- Create a working directory for developing your Node.js module. Name this directory `awsnodesample`. Note that the directory must be created in a location that can be updated by applications. For example, in Windows, do not create the directory under "C:\Program Files".
- Install Node.js. For more information, see the [Node.js website](https://nodejs.org/en/download/current/). You can find downloads of the current and LTS versions of Node.js for a variety of operating systems at <https://nodejs.org/en/download/current/>.

Contents

- [Step 1: Install the SDK and Dependencies](#)

- [Step 2: Configure Your Credentials](#)
- [Step 3: Create the Package JSON for the Project](#)
- [Step 4: Write the Node.js Code](#)
- [Step 5: Run the Sample](#)

Step 1: Install the SDK and Dependencies

You install the SDK for JavaScript package using [npm \(the Node.js package manager\)](#).

From the `awsnodesample` directory in the package, type the following at the command line.

```
npm install aws-sdk
```

This command installs the SDK for JavaScript in your project, and updates `package.json` to list the SDK as a project dependency. You can find information about this package by searching for "aws-sdk" on the [npm website](#).

Next, install the `uuid` module to the project by typing the following at the command line, which installs the module and updates `package.json`. For more information about `uuid`, see the module's page at <https://www.npmjs.com/package/uuid>.

```
npm install uuid
```

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

For more information on 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](#).

Step 2: Configure Your Credentials

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 [SDK authentication with AWS](#).

To hold this information, we recommend you create a shared credentials file. To learn how, see [Loading Credentials in Node.js from the Shared Credentials File](#). Your credentials file should resemble the following example.

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

You can determine whether you have set your credentials correctly by executing the following code with Node.js:

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

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

Similarly, if you have set your region correctly in your config file, you can display that value by setting the `AWS_SDK_LOAD_CONFIG` environment variable to any value and using the following code:

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

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

Step 3: Create the Package JSON for the Project

After you create the `awsnodesample` project directory, you create and add a `package.json` file for holding the metadata for your Node.js project. For details about using `package.json` in a Node.js project, see [Creating a package.json file](#).

In the project directory, create a new file named `package.json`. Then add this JSON to the file.

```
{
  "dependencies": {},
  "name": "aws-nodejs-sample",
```

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

Save the file. As you install the modules you need, the dependencies portion of the file will be completed. You can find a JSON file that shows an example of these dependencies [here on GitHub](#).

Step 4: Write the Node.js Code

Create a new file named `sample.js` to contain the example code. Begin by adding the `require` function calls to include the SDK for JavaScript and `uuid` modules so that they are available for you to use.

Build a unique bucket name that is used to create an Amazon S3 bucket by appending a unique ID value to a recognizable prefix, in this case `'node-sdk-sample-'`. You generate the unique ID by calling the `uuid` module. Then create a name for the Key parameter used to upload an object to the bucket.

Create a promise object to call the `createBucket` method of the `AWS.S3` service object. On a successful response, create the parameters needed to upload text to the newly created bucket. Using another promise, call the `putObject` method to upload the text object to the bucket.

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

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

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



```
.createBucket({ Bucket: bucketName })
.promise();

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

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

Step 5: Run the Sample

Type the following command to run the sample.

```
node sample.js
```

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

Setting Up the SDK for JavaScript

The topics in this section explain how to install the SDK for JavaScript for use in web browsers and with Node.js. It also shows how to load the SDK so you can access the web services supported by the SDK.

Note

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

Topics

- [Prerequisites](#)
- [Installing the SDK for JavaScript](#)
- [Loading the SDK for JavaScript](#)
- [Upgrading the SDK for JavaScript from Version 1](#)

Prerequisites

Before you use the AWS SDK for JavaScript, determine whether your code needs to run in Node.js or web browsers. After that, do the following:

- For Node.js, install Node.js on your servers if it is not already installed.
- For web browsers, identify the browser versions you need to support.

Topics

- [Setting Up an AWS Node.js Environment](#)
- [Web Browsers Supported](#)

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 pre-installed and 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 version of Node.js (32-bit or 64-bit) pre-installed.
- 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 [Tutorial: Setting Up Node.js on an Amazon EC2 Instance](#).
- Create a serverless environment using AWS Lambda to run Node.js as a Lambda function. For more information about using Node.js within a Lambda function, see [Programming Model \(Node.js\)](#) in the *AWS Lambda Developer Guide*.
- Deploy your Node.js application to AWS Elastic Beanstalk. For more information on using Node.js with Elastic Beanstalk, see [Deploying Node.js Applications to AWS Elastic Beanstalk](#) in the *AWS Elastic Beanstalk Developer Guide*.

Web Browsers Supported

The SDK for JavaScript supports all modern web browsers, including these minimum versions:

Browser	Version
Google Chrome	28.0+
Mozilla Firefox	26.0+
Opera	17.0+
Microsoft Edge	25.10+
Windows Internet Explorer	N/A
Apple Safari	5+
Android Browser	4.3+

Note

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

Installing the SDK for JavaScript

Whether and how you install the AWS SDK for JavaScript depends whether the code executes in Node.js modules or browser scripts.

Not all services are immediately available in the SDK. To find out which services are currently supported by the AWS SDK for JavaScript, see <https://github.com/aws/aws-sdk-js/blob/master/SERVICES.md>

Node

The preferred way to install the AWS SDK for JavaScript for Node.js is to use [npm, the Node.js package manager](#). To do so, type this at the command line.

```
npm install aws-sdk
```

In the event you see this error message:

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

Type these commands at the command line:

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

Browser

You don't have to install the SDK to use it in browser scripts. You can load the hosted SDK package directly from Amazon Web Services with a script in your HTML pages. The hosted SDK package supports the subset of AWS services that enforce cross-origin resource sharing (CORS). For more information, see [Loading the SDK for JavaScript](#).

You can create a custom build of the SDK in which you select the specific web services and versions that you want to use. You then download your custom SDK package for local

development and host it for your application to use. For more information about creating a custom build of the SDK, see [Building the SDK for Browsers](#).

You can download minified and non-minified distributable versions of the current AWS SDK for JavaScript from GitHub at:

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

Installing Using Bower

[Bower](#) is a package manager for the web. After you install Bower, you can use it to install the SDK. To install the SDK using Bower, type the following into a terminal window:

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

Loading the SDK for JavaScript

How you load the SDK for JavaScript depends on whether you are loading it to run in a web browser or in Node.js.

Not all services are immediately available in the SDK. To find out which services are currently supported by the AWS SDK for JavaScript, see <https://github.com/aws/aws-sdk-js/blob/master/SERVICES.md>

Node.js

After you install the SDK, you can load the AWS package in your node application using `require`.

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

React Native

To use the SDK in a React Native project, first install the SDK using npm:

```
npm install aws-sdk
```

In your application, reference the React Native compatible version of the SDK with the following code:

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

Browser

The quickest way to get started with the SDK is to load the hosted SDK package directly from Amazon Web Services. To do this, add a `<script>` element to your HTML pages in the following form:

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

To find the current `SDK_VERSION_NUMBER`, see the API Reference for the SDK for JavaScript at [AWS SDK for JavaScript API Reference Guide](#).

After the SDK loads in your page, the SDK is available from the global variable `AWS` (or `window.AWS`).

If you bundle your code and module dependencies using [browserify](#), you load the SDK using `require`, just as you do in Node.js.

Upgrading the SDK for JavaScript from Version 1

The following notes help you upgrade the SDK for JavaScript from version 1 to version 2.

Automatic Conversion of Base64 and Timestamp Types on Input/Output

The SDK now automatically encodes and decodes base64-encoded values, as well as timestamp values, on the user's behalf. This change affects any operation where base64 or timestamp values were sent by a request or returned in a response that allows for base64-encoded values.

User code that previously converted base64 is no longer required. Values encoded as base64 are now returned as buffer objects from server responses and can also be passed as buffer input. For example, the following version 1 `SQS.sendMessage` parameters:

```
var params = {  
  MessageBody: 'Some Message',  
  MessageAttributes: {  
    attrName: {
```

```
        DataType: 'Binary',
        BinaryValue: new Buffer('example text').toString('base64')
    }
}
};
```

Can be rewritten as follows.

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

Here is how the message is read.

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

Moved response.data.RequestId to response.requestId

The SDK now stores request IDs for all services in a consistent place on the response object, rather than inside the `response.data` property. This improves consistency across services that expose request IDs in different ways. This is also a breaking change that renames the `response.data.RequestId` property to `response.requestId` (`this.requestId` inside a callback function).

In your code, change the following:

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

To the following:

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

Exposed Wrapper Elements

If you use `AWS.ElastiCache`, `AWS.RDS`, or `AWS.Redshift`, you must access the response through the top-level output property in the response for some operations.

For example, the `RDS.describeEngineDefaultParameters` method used to return the following.

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

It now returns the following.

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

The list of affected operations for each service are shown in the following table.

Client Class	Operations
AWS.ElastiCache	authorizeCacheSecurityGroupIngress
	createCacheCluster
	createCacheParameterGroup
	createCacheSecurityGroup
	createCacheSubnetGroup
	createReplicationGroup
	deleteCacheCluster
	deleteReplicationGroup
	describeEngineDefaultParameters

Client Class	Operations
	<code>modifyCacheCluster</code>
	<code>modifyCacheSubnetGroup</code>
	<code>modifyReplicationGroup</code>
	<code>purchaseReservedCacheNodesOffering</code>
	<code>rebootCacheCluster</code>
	<code>revokeCacheSecurityGroupIngress</code>

Client Class	Operations
<code>AWS.RDS</code>	<code>addSourceIdentifierToSubscription</code> <code>authorizeDBSecurityGroupIngress</code> <code>copyDBSnapshot</code> <code>createDBInstance</code> <code>createDBInstanceReadReplica</code> <code>createDBParameterGroup</code> <code>createDBSecurityGroup</code> <code>createDBSnapshot</code> <code>createDBSubnetGroup</code> <code>createEventSubscription</code> <code>createOptionGroup</code> <code>deleteDBInstance</code> <code>deleteDBSnapshot</code> <code>deleteEventSubscription</code> <code>describeEngineDefaultParameters</code> <code>modifyDBInstance</code> <code>modifyDBSubnetGroup</code> <code>modifyEventSubscription</code> <code>modifyOptionGroup</code> <code>promoteReadReplica</code>

Client Class	Operations
	<code>purchaseReservedDBInstancesOffering</code>
	<code>rebootDBInstance</code>
	<code>removeSourceIdentifierFromSubscription</code>
	<code>restoreDBInstanceFromDBSnapshot</code>
	<code>restoreDBInstanceToPointInTime</code>
	<code>revokeDBSecurityGroupIngress</code>

Client Class	Operations
AWS.Redshift	<code>authorizeClusterSecurityGroupIngress</code> <code>authorizeSnapshotAccess</code> <code>copyClusterSnapshot</code> <code>createCluster</code> <code>createClusterParameterGroup</code> <code>createClusterSecurityGroup</code> <code>createClusterSnapshot</code> <code>createClusterSubnetGroup</code> <code>createEventSubscription</code> <code>createHsmClientCertificate</code> <code>createHsmConfiguration</code> <code>deleteCluster</code> <code>deleteClusterSnapshot</code> <code>describeDefaultClusterParameters</code> <code>disableSnapshotCopy</code> <code>enableSnapshotCopy</code> <code>modifyCluster</code> <code>modifyClusterSubnetGroup</code> <code>modifyEventSubscription</code>

Client Class	Operations
	<code>modifySnapshotCopyRetentionPeriod</code>
	<code>purchaseReservedNodeOffering</code>
	<code>rebootCluster</code>
	<code>restoreFromClusterSnapshot</code>
	<code>revokeClusterSecurityGroupIngress</code>
	<code>revokeSnapshotAccess</code>
	<code>rotateEncryptionKey</code>

Dropped Client Properties

The `.Client` and `.client` properties have been removed from service objects. If you use the `.Client` property on a service class or a `.client` property on a service object instance, remove these properties from your code.

The following code used with version 1 of the SDK for JavaScript:

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

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

Should be changed to the following code.

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

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

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

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

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

Topics

- [Using the Global Configuration Object](#)
- [Setting the AWS Region](#)
- [Specifying Custom Endpoints](#)
- [SDK authentication with AWS](#)
- [Setting Credentials](#)
- [Locking API Versions](#)
- [Node.js Considerations](#)
- [Browser Script Considerations](#)
- [Bundling Applications with Webpack](#)

Using the Global Configuration Object

There are two ways to configure the SDK:

- Set the global configuration using `AWS.Config`.

- Pass extra configuration information to a service object.

Setting global configuration with `AWS.Config` is often easier to get started, but service-level configuration can provide more control over individual services. The global configuration specified by `AWS.Config` provides default settings for service objects that you create subsequently, simplifying their configuration. However, you can update the configuration of individual service objects when your needs vary from the global configuration.

Setting Global Configuration

After you load the `aws-sdk` package in your code you can use the `AWS` global variable to access the SDK's classes and interact with individual services. The SDK includes a global configuration object, `AWS.Config`, that you can use to specify the SDK configuration settings required by your application.

Configure the SDK by setting `AWS.Config` properties according to your application needs. The following table summarizes `AWS.Config` properties commonly used to set the configuration of the SDK.

Configuration Options	Description
<code>credentials</code>	Required. Specifies the credentials used to determine access to services and resources.
<code>region</code>	Required. Specifies the Region in which requests for services are made.
<code>maxRetries</code>	Optional. Specifies the maximum number of times a given request is retried.
<code>logger</code>	Optional. Specifies a logger object to which debugging information is written.
<code>update</code>	Optional. Updates the current configuration with new values.

For more information about the configuration object, see [Class: `AWS.Config`](#) in the API Reference.

Global Configuration Examples

You must set the Region and the credentials in `AWS.Config`. You can set these properties as part of the `AWS.Config` constructor, as shown in the following browser script example:

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

You can also set these properties after creating `AWS.Config` using the update method, as shown in the following example that updates the Region:

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

You can get your default credentials by calling the static `getCredentials` method of `AWS.config`:

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

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

Similarly, if you have set your region correctly in your config file, you get that value by setting the `AWS_SDK_LOAD_CONFIG` environment variable is set to any value and calling the static `region` property of `AWS.config`:

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

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


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. When you set configuration values on a service object, the constructor takes all of the configuration values used by `AWS.Config`, including credentials.

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

You can also set configuration values specific to a service when configuring the SDK with `AWS.Config`. The global configuration object supports many service-specific configuration options. For more information about service-specific configuration, see [Class: `AWS.Config`](#) in the AWS SDK for JavaScript API Reference.

Immutable Configuration Data

Global configuration changes apply to requests for all newly created service objects. Newly created service objects are configured with the current global configuration data first and then any local configuration options. Updates you make to the global `AWS.config` object don't apply to previously created service objects.

Existing service objects must be manually updated with new configuration data or you must create and use a new service object that has the new configuration data. The following example creates a new Amazon S3 service object with new configuration data:

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

Setting the AWS Region

A 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 configuring

the SDK for JavaScript so that the SDK accesses the resources 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 Region using an environment variable, a shared config file, or the global configuration object.

In a Client Class Constructor

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

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

Using the Global Configuration Object

To set the Region in your JavaScript code, update the `AWS.Config` global configuration object as shown here.

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

For more information about current Regions and available services in each Region, see [AWS Regions and Endpoints](#) in the *AWS General Reference*.

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 Region and other configuration settings in a shared file named `config` that is used by SDKs. If the `AWS_SDK_LOAD_CONFIG` environment variable has been set to any 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-east-1
  output=json
```

For more information about using shared config and credentials files, see [Loading Credentials in Node.js from the Shared Credentials File](#) or [Configuration and Credential Files](#) in the *AWS Command Line Interface User Guide*.

Order of Precedence for Setting the Region

The order of precedence for Region setting is as follows:

- If a Region is passed to a client class constructor, that Region is used. If not, then...
- If a Region is set on the global configuration object, that Region is used. If not, then...
- If the AWS_REGION environment variable is a [truthy](#) value, that Region is used. If not, then...
- If the AMAZON_REGION environment variable is a truthy value, that Region is used. If not, then...
- If the AWS_SDK_LOAD_CONFIG environment variable is set to any value and the shared credentials file (~/.aws/credentials or the path indicated by AWS_SHARED_CREDENTIALS_FILE) contains a Region for the configured profile, that Region is used. If not, then...
- If the AWS_SDK_LOAD_CONFIG environment variable is set to any value and the config file (~/.aws/config or the path indicated by AWS_CONFIG_FILE) contains a Region for the configured profile, that Region is used.

Specifying Custom Endpoints

Calls to API methods in the SDK for JavaScript are made to service endpoint URIs. By default, these endpoints are built from the Region you have configured for your code. However, there are situations in which you need to specify a custom endpoint for your API calls.

Endpoint String Format

Endpoint values should be a string in the format:

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

Endpoints for the ap-northeast-3 Region

The ap-northeast-3 Region in Japan is not returned by Region enumeration APIs, such as [EC2.describeRegions](#). To define endpoints for this Region, follow the format described previously. So the Amazon EC2 endpoint for this Region would be

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

Endpoints for MediaConvert

You need to create a custom endpoint to use with MediaConvert. Each customer account is assigned its own endpoint, which you must use. Here is an example of how to use a custom endpoint with MediaConvert.

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

var getJobParams = {Id: 'job_ID'};

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

To get your account API endpoint, see [MediaConvert.describeEndpoints](#) in the API Reference.

Make sure you specify the same Region in your code as the Region in the custom endpoint URI. A mismatch between the Region setting and the custom endpoint URI can cause API calls to fail.

For more information on MediaConvert, see the [AWS.MediaConvert](#) class in the API Reference or the [AWS Elemental MediaConvert User Guide](#).

SDK authentication with AWS

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

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

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

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

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

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

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

The SDK for JavaScript does not need additional packages (such as SS0 and SS00IDC) to be added to your application to use IAM Identity Center authentication.

Start an AWS access portal session

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

```
aws sso login
```

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

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

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

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

Note

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

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

More authentication information

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

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

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

Setting Credentials

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

Whether running in a web browser or in a Node.js server, your JavaScript code must obtain valid credentials before it can access services through the API. Credentials can be set globally on the configuration object, using `AWS.Config`, or 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 on 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 sensitive information.

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](#)
- [Setting Credentials in a Web Browser](#)

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.

Here are the ways 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

If more than one credential source is available to the SDK, the default precedence of selection is as follows:

1. Credentials that are explicitly set through the service-client constructor

2. Environment variables
3. The shared credentials file
4. Credentials loaded from the ECS credentials provider (if applicable)
5. Credentials that are obtained by using a credential process specified in the shared AWS config file or the shared credentials file. For more information, see [the section called “Credentials using a Configured Credential Process”](#).
6. Credentials loaded from AWS IAM using the credentials provider of the Amazon EC2 instance (if configured in the instance metadata)

For more information, see [Class: `AWS.Credentials`](#) and [Class: `AWS.CredentialProviderChain`](#) in the API reference.

Warning

While it is possible to do so, we do not 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](#)
- [Loading Credentials for a Node.js Lambda Function](#)
- [Loading Credentials in Node.js from the Shared Credentials File](#)
- [Loading Credentials in Node.js from Environment Variables](#)
- [Loading Credentials in Node.js from a JSON File](#)
- [Loading Credentials in Node.js using a Configured Credential Process](#)

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 on adding IAM roles to an Amazon EC2 instance, see [Using IAM roles for Amazon EC2 instances](#) in the *AWS SDKs and Tools Reference Guide*.

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 do not 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 [SDK authentication with AWS](#). 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:

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

For an example showing this file being used, see [Getting Started in Node.js](#).

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 above, 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 afterwards (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 JavaScript file that uses the SDK, such as for example, a file named `script.js`.

```
$ node script.js
```

You can also explicitly select the profile used by the SDK, either by setting `process.env.AWS_PROFILE` before loading the SDK, or by selecting the credential provider as shown in the following example:

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

```
AWS.config.credentials = credentials;
```

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, eliminating 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`

For more details on setting environment variables, see [Environment variables support](#) in the *AWS SDKs and Tools Reference Guide*.

Loading Credentials in Node.js from a JSON File

You can load configuration and credentials from a JSON document on disk using `AWS.config.loadFromPath`. The path specified is relative to the current working directory of your process. For example, to load credentials from a 'config.json' file with the following content:

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

Then use the following code:

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

Note

Loading configuration data from a JSON document resets all existing configuration data. Add additional configuration data after using this technique. Loading credentials from a JSON document is not supported in browser scripts.

Loading Credentials in Node.js using a Configured Credential Process

You can source credentials by using a method that isn't built into the SDK. To do this, specify a credential process in the shared AWS config file or the shared credentials file. If the `AWS_SDK_LOAD_CONFIG` environment variable is set to any value, the SDK will prefer the process specified in the config file over the process specified in the credentials file (if any).

For details about specifying a credential process in the shared AWS config file or the shared credentials file, see the *AWS CLI Command Reference*, specifically the information about [Sourcing Credentials From External Processes](#).

For information about using the `AWS_SDK_LOAD_CONFIG` environment variable, see [the section called "Using a Shared Config File"](#) in this document.

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 coded 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](#)
- [Using Web Federated Identity to Authenticate Users](#)
- [Web Federated Identity Examples](#)

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 object, `AWS.CognitoIdentityCredentials`. 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 with the `AWS.CognitoIdentityCredentials` object. 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 `AWS.CognitoIdentityCredentials`. Create and associate both authenticated and unauthenticated IAM roles for your identity pool.

Unauthenticated users do not 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 with identity providers attached, you can use `AWS.CognitoIdentityCredentials` to authenticate users. To configure your application credentials to use `AWS.CognitoIdentityCredentials`, set the `credentials` property of either `AWS.Config` or a per-service configuration. The following example uses `AWS.Config`:

```
AWS.config.credentials = new AWS.CognitoIdentityCredentials({
  IdentityPoolId: 'us-east-1:1699ebc0-7900-4099-b910-2df94f52a030',
  Logins: { // optional tokens, used for authenticated login
    'graph.facebook.com': 'FBTOKEN',
    'www.amazon.com': 'AMAZONTOKEN',
    'accounts.google.com': 'GOOGLETOKEN'
  }
})
```

```
});
```

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 Facebook is one of your identity providers, you might use the `FB.login` function from the [Facebook SDK](#) to get an identity provider token:

```
FB.login(function (response) {  
  if (response.authResponse) { // logged in  
    AWS.config.credentials = new AWS.CognitoIdentityCredentials({  
      IdentityPoolId: 'us-east-1:1699ebc0-7900-4099-b910-2df94f52a030',  
      Logins: {  
        'graph.facebook.com': response.authResponse.accessToken  
      }  
    });  
  
    s3 = new AWS.S3; // we can now create our service object  
  
    console.log('You are now logged in.');
```

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

```
}  
});
```

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 configuration might look like the following:

```
// set the default config object  
var creds = new AWS.CognitoIdentityCredentials({
```

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

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;  
}
```

You can also Create `CognitoIdentityCredentials` object. If you do, you must reset the credentials properties of existing service objects you created. Service objects read from the global configuration only on object initialization.

For more information about the `CognitoIdentityCredentials` object, see [AWS.CognitoIdentityCredentials](#) in the AWS SDK for JavaScript API Reference.

Using Web Federated Identity to Authenticate Users

You can directly configure individual identity providers to access AWS resources using web identity federation. AWS currently supports authenticating users using web identity federation through several identity providers:

- [Login with Amazon](#)
- [Facebook Login](#)
- [Google Sign-in](#)

You must first register your application with the providers that your application supports. Next, create an IAM role and set up permissions for it. The IAM role you create is then used to grant the permissions you configured for it through the respective identity provider. For example, you can

set up a role that allows users who logged in through Facebook to have read access to a specific Amazon S3 bucket you control.

After you have both an IAM role with configured privileges and an application registered with your chosen identity providers, you can set up the SDK to get credentials for the IAM role using helper code, as follows:

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

The value in the `ProviderId` parameter depends on the specified identity provider. The value in the `WebIdentityToken` parameter is the access token retrieved from a successful login with the identity provider. For more information on how to configure and retrieve access tokens for each identity provider, see the documentation for the identity provider.

Step 1: Registering with Identity Providers

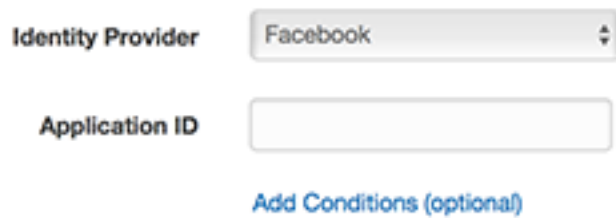
To begin, register an application with the identity providers you choose to support. You will be asked to provide information that identifies your application and possibly its author. This ensures that the identity providers know who is receiving their user information. In each case, the identity provider will issue an application ID that you use to configure user roles.

Step 2: Creating an IAM Role for an Identity Provider

After you obtain the application ID from an identity provider, go to the IAM console at <https://console.aws.amazon.com/iam/> to create a new IAM role.

To create an IAM role for an identity provider

1. Go to the **Roles** section of the console and then choose **Create New Role**.
2. Type a name for the new role that helps you keep track of its use, such as **facebookIdentity**, and then choose **Next Step**.
3. In **Select Role Type**, choose **Role for Identity Provider Access**.
4. For **Grant access to web identity providers**, choose **Select**.
5. From the **Identity Provider** list, choose the identity provider that you want to use for this IAM role.



Identity Provider: Facebook

Application ID:

[Add Conditions \(optional\)](#)

6. Type the application ID provided by the identity provider in **Application ID** and then choose **Next Step**.
7. Configure permissions for the resources you want to expose, allowing access to specific operations on specific resources. For more information about IAM permissions, see [Overview of AWS IAM Permissions](#) in the *IAM User Guide*. Review and, if needed, customize the role's trust relationship, and then choose **Next Step**.
8. Attach additional policies you need and then choose **Next Step**. For more information about IAM policies, see [Overview of IAM Policies](#) in the *IAM User Guide*.
9. Review the new role and then choose **Create Role**.

You can provide other constraints to the role, such as scoping it to specific user IDs. If the role grants write permissions to your resources, make sure you correctly scope the role to users with the correct privileges, otherwise any user with an Amazon, Facebook, or Google identity will be able to modify resources in your application.

For more information on using web identity federation in IAM, see [About Web Identity Federation](#) in the *IAM User Guide*.

Step 3: Obtaining a Provider Access Token After Login

Set up the login action for your application by using the identity provider's SDK. You can download and install a JavaScript SDK from the identity provider that enables user login, using either OAuth or OpenID. For information on how to download and set up the SDK code in your application, see the SDK documentation for your identity provider:

- [Login with Amazon](#)
- [Facebook Login](#)
- [Google Sign-in](#)

Step 4: Obtaining Temporary Credentials

After your application, roles, and resource permissions are configured, add the code to your application to obtain temporary credentials. These credentials are provided through the AWS Security Token Service using web identity federation. Users log in to the identity provider, which returns an access token. Set up the `AWS.WebIdentityCredentials` object using the ARN for the IAM role you created for this identity provider:

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

Service objects that are created subsequently will have the proper credentials. Objects created before setting the `AWS.config.credentials` property won't have the current credentials.

You can also create `AWS.WebIdentityCredentials` before retrieving the access token. This allows you to create service objects that depend on credentials before loading the access token. To do this, create the credentials object without the `WebIdentityToken` parameter:

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

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

Then set `WebIdentityToken` in the callback from the identity provider SDK that contains the access token:

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

Web Federated Identity Examples

Here are a few examples of using web federated identity to obtain credentials in browser JavaScript. These examples must be run from an `http://` or `https://` host scheme to ensure the identity provider can redirect to your application.

Login with Amazon Example

The following code shows how to use Login with Amazon as an identity provider.

```
<a href="#" id="login">
  
</a>
<div id="amazon-root"></div>
<script type="text/javascript">
  var s3 = null;
  var clientId = 'amzn1.application-oa2-client.1234567890abcdef'; // client ID
  var roleArn = 'arn:aws:iam::<AWS_ACCOUNT_ID>:role/<WEB_IDENTITY_ROLE_NAME>';

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

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

          s3 = new AWS.S3();

          console.log('You are now logged in.');
```

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

```
        }
      });
    };
  };

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

```
</script>
```

Facebook Login Example

The following code shows how to use Facebook Login as an identity provider:

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

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

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

        s3 = new AWS.S3;

        console.log('You are now logged in.');
```

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

```
      }
    });
  };
};

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

```
</script>
```

Google+ Sign-in Example

The following code shows how to use Google+ Sign-in as an identity provider. The access token used for web identity federation from Google is stored in `response.id_token` instead of `access_token` like other identity providers.

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

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

      s3 = new AWS.S3();

      console.log('You are now logged in.');
```

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

```
    }
  }

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

```
})();  
</script>
```

Locking API Versions

AWS services have API version numbers to keep track of API compatibility. API versions in AWS services are identified by a YYYY-mm-dd formatted date string. For example, the current API version for Amazon S3 is 2006-03-01.

We recommend locking the API version for a service if you rely on it in production code. This can isolate your applications from service changes resulting from updates to the SDK. If you don't specify an API version when creating service objects, the SDK uses the latest API version by default. This could cause your application to reference an updated API with changes that negatively impact your application.

To lock the API version that you use for a service, pass the `apiVersion` parameter when constructing the service object. In the following example, a newly created `AWS.DynamoDB` service object is locked to the 2011-12-05 API version:

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

You can globally configure a set of service API versions by specifying the `apiVersions` parameter in `AWS.Config`. For example, to set specific versions of the DynamoDB and Amazon EC2 APIs along with the current Amazon Redshift API, set `apiVersions` as follows:

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

Getting API Versions

To get the API version for a service, see the *Locking the API Version* section on the service's reference page, such as <https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/S3.html> for Amazon S3.

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.

```
var http = require('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
var fs = require('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](#).

One example of an npm package you can use with the AWS SDK for JavaScript is `browserify`. For details, see [Building the SDK as a Dependency with Browserify](#). Another example is `webpack`. For details, see [Bundling Applications with Webpack](#).

Topics

- [Configuring maxSockets in Node.js](#)
- [Reusing Connections with Keep-Alive in Node.js](#)
- [Configuring Proxies for Node.js](#)
- [Registering Certificate Bundles in Node.js](#)

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 all service objects you create. This example allows up to 25 concurrent connections to each service endpoint.

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

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

The same can be done per service.

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

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

When using the default of `https`, the SDK takes the `maxSockets` value from the `globalAgent`. If the `maxSockets` value is not defined or is `Infinity`, 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

By default, 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, you can reuse an existing connection.

For short-lived operations, such as 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.

The easiest way to configure SDK for JavaScript to reuse TCP connections is to set the `AWS_NODEJS_CONNECTION_REUSE_ENABLED` environment variable to 1. This feature was added in the [2.463.0](#) release.

Alternatively, you can set the `keepAlive` property of an HTTP or HTTPS agent set to `true`, as shown in the following example.

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

```
// Infinity is read as 50 sockets
  maxSockets: Infinity
});

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

The following example shows how to set `keepAlive` for just a DynamoDB client:

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

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

If `keepAlive` is enabled, you can also set the initial delay for TCP Keep-Alive packets with `keepAliveMsecs`, which by default is 1000ms. 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, such as [proxy-agent](#). To install `proxy-agent`, type the following at the command line.

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

If you decide to use a different proxy, first follow the installation and configuration instructions for that proxy. To use this or another third-party proxy in your application, you must set the `httpOptions` property of `AWS.Config` to specify the proxy you choose. This example shows `proxy-agent`.

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

For more information about other proxy libraries, see [npm, the Node.js package manager](#).

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 to update the SDK configuration.

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

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

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](#)

- [Cross-Origin Resource Sharing \(CORS\)](#)

Building the SDK for Browsers

The SDK for JavaScript is provided as a JavaScript file with support included for a default set of services. This file is typically loaded into browser scripts using a `<script>` tag that references the hosted SDK package. However, you may need support for services other than the default set or otherwise need to customize the SDK.

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.

Topics

- [Using the SDK Builder to Build the SDK for JavaScript](#)
- [Using the CLI to Build the SDK for JavaScript](#)
- [Building Specific Services and API Versions](#)
- [Building the SDK as a Dependency with Browserify](#)

Using the SDK Builder to Build the SDK for JavaScript

The easiest way to create your own build of the AWS SDK for JavaScript is to use the SDK builder web application at <https://sdk.amazonaws.com/builder/js>. Use the SDK builder to specify services, and their API versions, to include in your build.

Choose **Select all services** or choose **Select default services** as a starting point from which you can add or remove services. Choose **Development** for more readable code or choose **Minified** to create a minified build to deploy. After you choose the services and versions to include, choose **Build** to build and download your custom SDK.

Using the CLI to Build the SDK for JavaScript

To build the SDK for JavaScript using the AWS CLI, you first need to clone the Git repository that contains the SDK source. You must have Git and Node.js installed on your computer.

First, clone the repository from GitHub and change directory into the directory:

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

After you clone the repository, download the dependency modules for both the SDK and build tool:

```
npm install
```

You can now build a packaged version of the SDK.

Building from the Command Line

The builder tool is in `dist-tools/browser-builder.js`. Run this script by typing:

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

This command builds the `aws-sdk.js` file. This file is uncompressed. By default this package includes only the standard set of services.

Minifying Build Output

To reduce the amount of data on the network, JavaScript files can be compressed through a process called *minification*. Minification strips comments, unnecessary spaces, and other characters that aid in human readability but that do not impact execution of the code. The builder tool can produce uncompressed or minified output. To minify your build output, set the `MINIFY` environment variable:

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

Building Specific Services and API Versions

You can select which services to build into the SDK. To select services, specify the service names, delimited by commas, as parameters. For example, to build only Amazon S3 and Amazon EC2, use the following command:

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

You can also select specific API versions of the services build by adding the version name after the service name. For example, to build both API versions of Amazon DynamoDB, use the following command:

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

Service identifiers and API versions are available in the service-specific configuration files at <https://github.com/aws/aws-sdk-js/tree/master/apis>.

Building All Services

You can build all services and API versions by including the `all` parameter:

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

Building Specific Services

To customize the selected set of services included in the build, pass the `AWS_SERVICES` environment variable to the Browserify command that contains the list of services you want. The following example builds the Amazon EC2, Amazon S3, and DynamoDB services.

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

Building the SDK as a Dependency with Browserify

Node.js has a module-based mechanism for including code and functionality from third-party developers. This modular approach is not natively supported by JavaScript running in web browsers. However, with a tool called Browserify, you can use the Node.js module approach and use modules written for Node.js in the browser. Browserify builds the module dependencies for a browser script into a single, self-contained JavaScript file that you can use in the browser.

You can build the SDK as a library dependency for any browser script by using Browserify. For example, the following Node.js code requires the SDK:

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

This example code can be compiled into a browser-compatible version using Browserify:

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

The application, including its SDK dependencies, is then made available in the browser through `browser-app.js`.

For more information about Browserify, see the [Browserify website](#).

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.

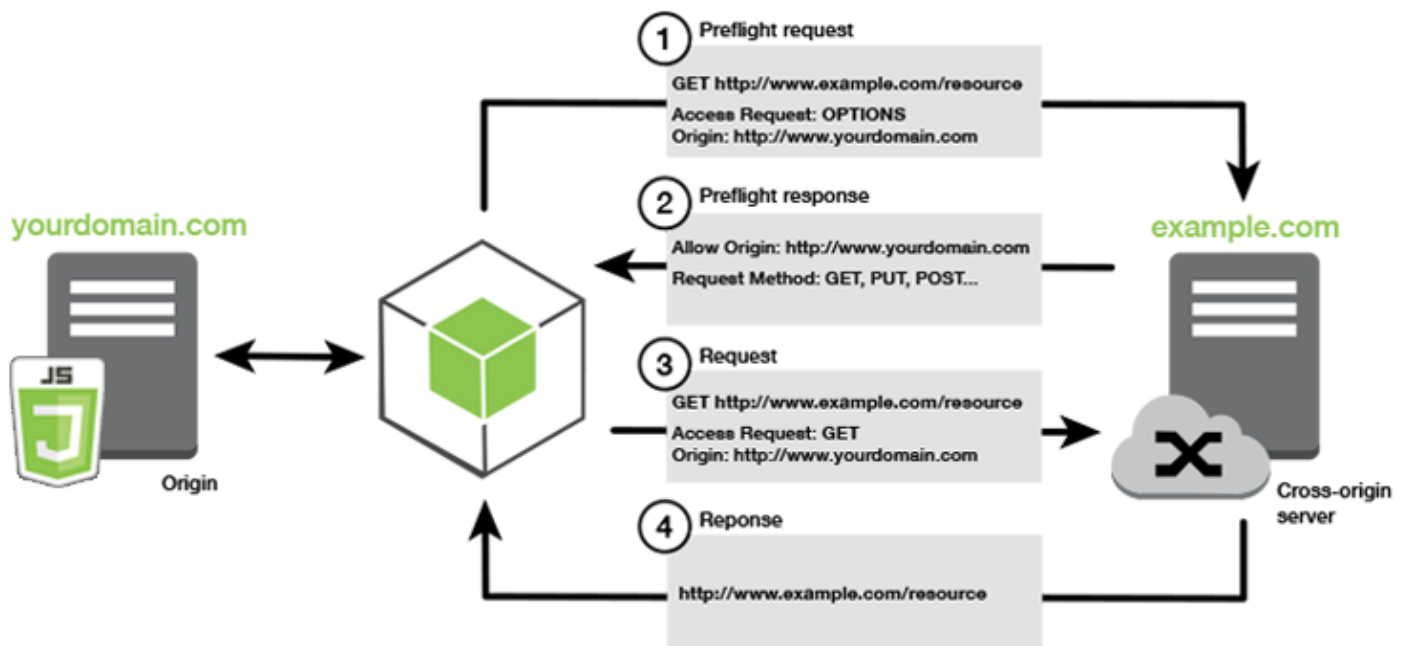
CORS determines whether to allow sharing of resources in a cross-origin request based on:

- The 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.



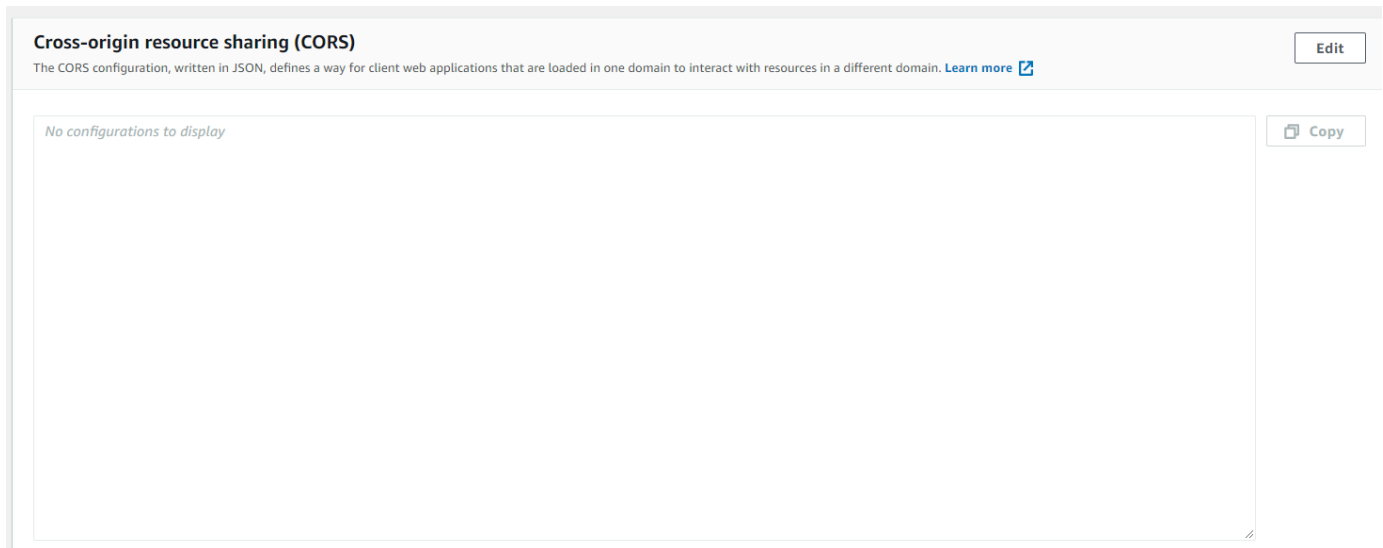
Is CORS Configuration Required

Amazon S3 buckets require CORS configuration before you can perform operations on them. In some JavaScript environments CORS may 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.

1. In the Amazon S3 console, choose the bucket you want to edit.
2. Select the **Permissions** tab, and scroll down to the **Cross-origin resource sharing (CORS)** panel.



3. Choose **Edit**, and type your CORS configuration in the **CORS Configuration Editor**, 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>`, which specifies domain origins that you allow to make cross-domain requests.
- `<AllowedMethod>`, which specifies a type of request you allow (GET, PUT, POST, DELETE, HEAD) in cross-domain requests.
- `<AllowedHeader>`, which specifies the headers allowed in a preflight request.

For sample 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 sample allows a user to view, add, remove, or update objects inside of a bucket from the domain `example.org`, though it is recommended 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, if you want to 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

Web applications in browser scripts or Node.js use of code modules 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 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 webpack
```

You may also need to install a webpack plugin that allows it to load JSON files. Type the following command to install the JSON loader plugin.

```
npm install json-loader
```

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. Here is an example of a `webpack.config.js` configuration file.

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

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.

The configuration in the `json-loader` plugin that was installed earlier specifies to webpack how to import JSON files. By default, webpack only supports JavaScript but uses loaders to add support

for importing other file types. Because the SDK for JavaScript makes extensive use of JSON files, webpack throws an error when generating the bundle if `json-loader` isn't included.

Running Webpack

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

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

Here is an example `package.json` 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": "^2.6.1"
  },
  "devDependencies": {
    "json-loader": "^0.5.4",
    "webpack": "^1.13.2"
  }
}
```

To build your application, type 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>AWS SDK with webpack</title>
  </head>
  <body>
    <div id="list"></div>
    <script src="bundle.js"></script>
  </body>
</html>
```

Importing Individual Services

One of the benefits of webpack is that it parses the dependencies in your code and bundles only the code your application needs. If you are using the SDK for JavaScript, bundling only the parts of the SDK actually used by your application can reduce the size of the webpack output considerably.

Consider the following example of the code used to create an Amazon S3 service object.

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

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

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

The `require()` function specifies the entire SDK. A webpack bundle generated with this code would include the full SDK but the full SDK is not required when only the Amazon S3 client class is used. The size of the bundle would be substantially smaller if only the portion of the SDK you require for the Amazon S3 service was included. Even setting the configuration doesn't require the full SDK because you can set the configuration data on the Amazon S3 service object.

Here is what the same code looks like when it includes only the Amazon S3 portion of the SDK.

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

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

Bundling for Node.js

You can use webpack to generate bundles that run in Node.js by specifying it 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, 'node.js')
  ],
  // Specify the output file containing our bundled code
  output: {
    path: __dirname,
    filename: 'bundle.js'
  },
  // Let webpack know to generate a Node.js bundle
  target: "node",
  module: {
    /**
     * Tell webpack how to load JSON files.
     * When webpack encounters a 'require()' statement
     * where a JSON file is being imported, it will use
     * the json-loader
     */
  }
```

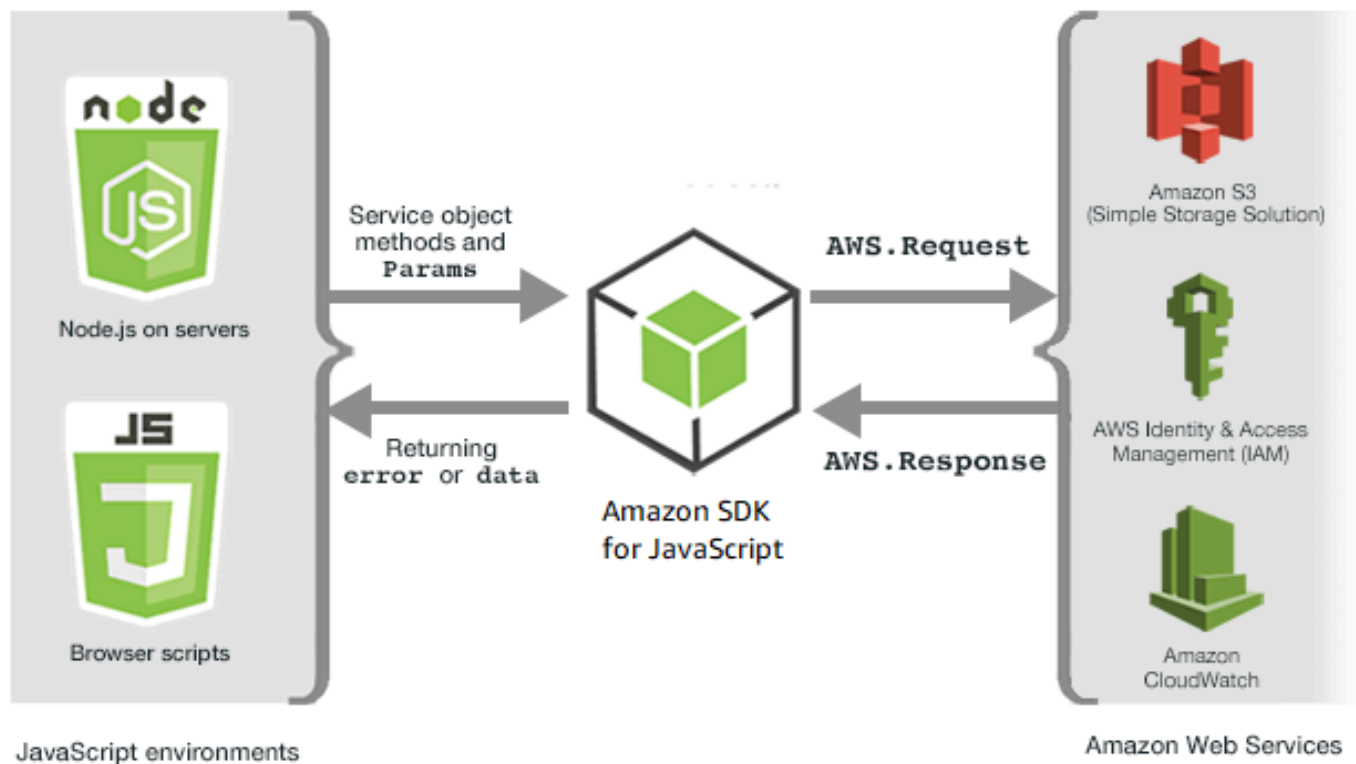


```
    */  
    loaders: [  
      {  
        test: /\.json$/,  
        loaders: ['json']  
      }  
    ]  
  }  
}
```

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 `AWS.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 is encapsulated in the SDK by the `AWS . Request` object. The response is encapsulated in the SDK by the `AWS . Response` object, which is provided to the requestor through one of several techniques, such as a callback function or a JavaScript promise.

Topics

- [Creating and Calling Service Objects](#)
- [Logging AWS SDK for JavaScript Calls](#)
- [Calling Services Asynchronously](#)
- [Using the Response Object](#)
- [Working with JSON](#)
- [Retry strategy in the AWS SDK for JavaScript v2](#)

Creating and Calling Service Objects

The JavaScript API supports most available AWS services. Each service class in the JavaScript API provides access to every API call in its service. For more information on 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 to your application using `require`, which provides support for all current services.

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

When using the SDK with browser JavaScript, you load the SDK package to your browser scripts using the AWS-hosted SDK package. To load the SDK package, add the following `<script>` element:

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

To find the current `SDK_VERSION_NUMBER`, see the API Reference for the SDK for JavaScript at [AWS SDK for JavaScript API Reference Guide](#).

The default hosted SDK package provides support for a subset of the available AWS services. For a list of the default services in the hosted SDK package for the browser, see [Supported Services](#) in the API Reference. You can use the SDK with other services if CORS security checking is disabled. In this case, you can build a custom version of the SDK to include the additional services you require. For more information on building a custom version of the SDK, see [Building the SDK for Browsers](#).

Requiring Individual Services

Requiring the SDK for JavaScript as shown previously includes the entire SDK into your code. Alternately, you can choose to require only the individual services used by your code. Consider the following code used to create an Amazon S3 service object.

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

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

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

In the previous example, the `require` function specifies the entire SDK. The amount of code to transport over the network as well as the memory overhead of your code would be substantially smaller if only the portion of the SDK you require for the Amazon S3 service was included. To require an individual service, call the `require` function as shown, including the service constructor in all lower case.

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

Here is what the code to create the previous Amazon S3 service object looks like when it includes only the Amazon S3 portion of the SDK.

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

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

You can still access the global AWS namespace without every service attached to it.

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

This is a useful technique when applying the same configuration across multiple individual services, for example to provide the same credentials to all services. Requiring individual services should reduce loading time and memory consumption in Node.js. When done along with a bundling tool such as Browserify or webpack, requiring individual services results in the SDK being a fraction of the full size. This helps with memory or disk-space constrained environments such as an IoT device or in a Lambda function.

Creating Service Objects

To access service features through the JavaScript API, you first create a *service object* through which you access a set of features provided by the underlying client class. Generally there is one client class provided for each service; however, some services divide access to their features among multiple client classes.

To use a feature, you must create an instance of the class that provides access to that feature. The following example shows creating a service object for DynamoDB from the `AWS.DynamoDB` client class.

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

By default, a service object is configured with the global settings also used to configure the SDK. However, you can configure a service object with runtime configuration data that is specific to that service object. Service-specific configuration data is applied after applying the global configuration settings.

In the following example, an Amazon EC2 service object is created with configuration for a specific Region but otherwise uses the global configuration.

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

In addition to supporting service-specific configuration applied to an individual service object, you can also apply service-specific configuration to all newly created service objects of a given class. For example, to configure all service objects created from the Amazon EC2 class to use the US West (Oregon) (us-west-2) Region, add the following to the `AWS.config` global configuration object.

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

Locking the API Version of a Service Object

You can lock a service object to a specific API version of a service by specifying the `apiVersion` option when creating the object. In the following example, a DynamoDB service object is created that is locked to a specific API version.

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

For more information about locking the API version of a service object, see [Locking API Versions](#).

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](#).

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

For more information about Amazon S3 parameters, see [Class: AWS.S3](#) in the API reference.

In addition, you can bind values to individual parameters when creating a service object using the `params` parameter. The value of the `params` parameter of service objects is a map that specifies one or more of the parameter values defined by the service object. The following example shows the `Bucket` parameter of an Amazon S3 service object being bound to a bucket named `amzn-s3-demo-bucket`.

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

By binding the service object to a bucket, the `s3bucket` service object treats the `amzn-s3-demo-bucket` parameter value as a default value that no longer needs to be specified for subsequent operations. Any bound parameter values are ignored when using the object for operations where the parameter value isn't applicable. You can override this bound parameter when making calls on the service object by specifying a new value.

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

```
// ...  
s3bucket.getObject({Bucket: 'amzn-s3-demo-bucket3', Key: 'key0therName'});
```

Details about available parameters for each method are found in the API reference.

Logging AWS SDK for JavaScript Calls

The AWS SDK for JavaScript is instrumented with a built-in logger so you can log API calls you make with the SDK for JavaScript.

To turn on the logger and print log entries in the console, add the following statement to your code.

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

Here is an example of the log output.

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

Using a Third-Party Logger

You can also use a third-party logger, provided it has `log()` or `write()` operations to write to a log file or server. You must install and set up your custom logger as instructed before you can use it with the SDK for JavaScript.

One such logger you can use in either browser scripts or in Node.js is `logplease`. In Node.js, you can configure `logplease` to write log entries to a log file. You can also use it with `webpack`.

When using a third-party logger, set all options before assigning the logger to `AWS.Config.logger`. For example, the following specifies an external log file and sets the log level for `logplease`

```
// Require AWS Node.js SDK  
const AWS = require('aws-sdk')  
// Require logplease  
const logplease = require('logplease');  
// Set external log file option  
logplease.setLogfile('debug.log');  
// Set log level
```

```
logplease.setLogLevel('DEBUG');  
// Create logger  
const logger = logplease.create('logger name');  
// Assign logger to SDK  
AWS.config.logger = logger;
```

For more information about logplease, see the [logplease Simple JavaScript Logger](#) on GitHub.

Calling Services Asynchronously

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

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

Topics

- [Managing Asynchronous Calls](#)
- [Using an Anonymous Callback Function](#)
- [Using a Request Object Event Listener](#)
- [Using async/await](#)
- [Using JavaScript Promises](#)

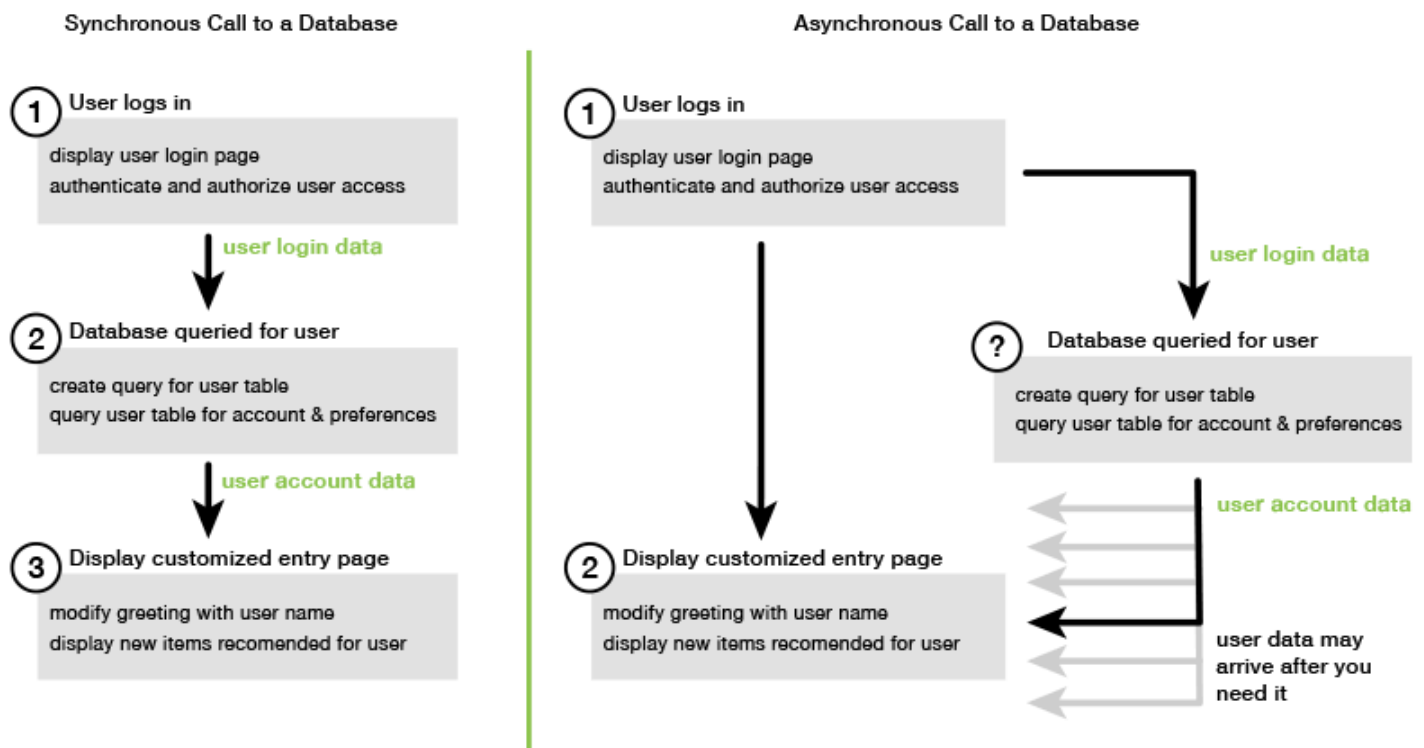
Managing Asynchronous Calls

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

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

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

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



Using an Anonymous Callback Function

Each service object method that creates an `AWS.Request` object can accept an anonymous callback function as the last parameter. The signature of this callback function is:

```
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.

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

Accessing the Request and Response Objects

Within the callback function, the JavaScript keyword `this` refers to the underlying `AWS.Response` object for most services. In the following example, the `httpResponse` property of an `AWS.Response` object is used within a callback function to log the raw response data and headers to help with debugging.

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

```
    console.log("Response data and headers: " + JSON.stringify(this.httpResponse));
  } else {
    console.log(data); // request succeeded
  }
});
```

In addition, because the `AWS.Response` object has a `Request` property that contains the `AWS.Request` that was sent by the original method call, you can also access the details of the request that was made.

Using a Request Object Event Listener

If you do not create and pass an anonymous callback function as a parameter when you call a service object method, the method call generates an `AWS.Request` object that must be manually sent using its `send` method.

To process the response, you must create an event listener for the `AWS.Request` object to register a callback function for the method call. The following example shows how to create the `AWS.Request` object for calling a service object method and the event listener for the successful return.

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

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

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

After the `send` method on the `AWS.Request` object is called, the event handler executes when the service object receives an `AWS.Response` object.

For more information about the `AWS.Request` object, see [Class: AWS.Request](#) in the API Reference. For more information about the `AWS.Response` object, see [Using the Response Object](#) or [Class: AWS.Response](#) in the API Reference.

Chaining Multiple Callbacks

You can register multiple callbacks on any request object. Multiple callbacks can be registered for different events or the same event. Also, you can chain callbacks as shown in the following example.

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

Request Object Completion Events

The `AWS.Request` object raises these completion events based on the response of each service operation method:

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

You can register a callback function in response to any of these events. For a complete list of all request object events, see [Class: `AWS.Request`](#) in the API Reference.

The success Event

The success event is raised upon a successful response received from the service object. Here is how you register a callback function for this event.

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

The response provides a data property that contains the serialized response data from the service. For example, the following call to the `listBuckets` method of the Amazon S3 service object

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

returns the response and then prints the following data property contents to the console.

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

The error Event

The error event is raised upon an error response received from the service object. Here is how you register a callback function for this event.

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

When the error event is raised, the value of the response's data property is `null` and the error property contains the error data. The associated error object is passed as the first parameter to the registered callback function. For example, the following code:

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

returns the error and then prints the following error data to the console.

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

The complete Event

The complete event is raised when a service object call has finished, regardless of whether the call results in success or error. Here is how you register a callback function for this event.

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

Use the `complete` event callback to handle any request cleanup that must execute regardless of success or error. If you use response data inside a callback for the `complete` event, first check the `response.data` or `response.error` properties before attempting to access either one, as shown in the following example.

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

Request Object HTTP Events

The `AWS.Request` object raises these HTTP events based on the response of each service operation method:

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

You can register a callback function in response to any of these events. For a complete list of all request object events, see [Class: `AWS.Request`](#) in the API Reference.

The `httpHeaders` Event

The `httpHeaders` event is raised when headers are sent by the remote server. Here is how you register a callback function for this event.

```
request.on('httpHeaders', function(statusCode, headers, response) {
```

```
// event handling code
});
```

The `statusCode` parameter to the callback function is the HTTP status code. The `headers` parameter contains the response headers.

The `httpData` Event

The `httpData` event is raised to stream response data packets from the service. Here is how you register a callback function for this event.

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

This event is typically used to receive large responses in chunks when loading the entire response into memory is not practical. This event has an additional `chunk` parameter that contains a portion of the actual data from the server.

If you register a callback for the `httpData` event, the `data` property of the response contains the entire serialized output for the request. You must remove the default `httpData` listener if you don't have the extra parsing and memory overhead for the built-in handlers.

The `httpUploadProgress` and `httpDownloadProgress` Events

The `httpUploadProgress` event is raised when the HTTP request has uploaded more data. Similarly, the `httpDownloadProgress` event is raised when the HTTP request has downloaded more data. Here is how you register a callback function for these events.

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

The `progress` parameter to the callback function contains an object with the loaded and total bytes of the request.

The `httpError` Event

The `httpError` event is raised when the HTTP request fails. Here is how you register a callback function for this event.

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

The error parameter to the callback function contains the error that was thrown.

The `httpDone` Event

The `httpDone` event is raised when the server finishes sending data. Here is how you register a callback function for this event.

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

Using `async/await`

You can use the `async/await` pattern in your calls to the AWS SDK for JavaScript. Most functions that take a callback do not return a promise. Since you only use `await` functions that return a promise, to use the `async/await` pattern you need to chain the `.promise()` method to the end of your call, and remove the callback.

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

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



```
run();
```

Note

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

Using JavaScript Promises

The `AWS.Request.promise` method provides a way to call a service operation and manage asynchronous flow instead of using callbacks. In Node.js and browser scripts, an `AWS.Request` object is returned when a service operation is called without a callback function. You can call the request's `send` method to make the service call.

However, `AWS.Request.promise` immediately starts the service call and returns a promise that is either fulfilled with the response data property or rejected with the response error property.

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

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

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

The next example returns a promise that's fulfilled with a data object, or rejected with an error object. Using promises, a single callback isn't responsible for detecting errors. Instead, the correct callback is called based on the success or failure of a request.

```
var s3 = new AWS.S3({apiVersion: '2006-03-01', region: 'us-west-2'});
var params = {
  Bucket: 'bucket',
  Key: 'example2.txt',
```

```
    Body: 'Uploaded text using the promise-based method!'
  };
  var putObjectPromise = s3.putObject(params).promise();
  putObjectPromise.then(function(data) {
    console.log('Success');
  }).catch(function(err) {
    console.log(err);
  });
});
```

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.

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

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

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

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 Other Promise Implementations

In addition to the native promise implementation in ECMAScript 2015, you can also use third-party promise libraries, including:

- [bluebird](#)

- [RSVP](#)
- [Q](#)

These optional promise libraries can be useful if you need your code to run in environments that don't support the native promise implementation in ECMAScript 5 and ECMAScript 2015.

To use a third-party promise library, set a promises dependency on the SDK by calling the `setPromisesDependency` method of the global configuration object. In browser scripts, make sure to load the third-party promise library before loading the SDK. In the following example, the SDK is configured to use the implementation in the bluebird promise library.

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

To return to using the native promise implementation of the JavaScript engine, call `setPromisesDependency` again, passing a `null` instead of a library name.

Using the Response Object

After a service object method has been called, it returns an `AWS.Response` object by passing it to your callback function. You access the contents of the response through the properties of the `AWS.Response` object. There are two properties of the `AWS.Response` object you use to access the contents of the response:

- `data` property
- `error` property

When using the standard callback mechanism, these two properties are provided as parameters on the anonymous callback function as shown in the following example.

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

Accessing Data Returned in the Response Object

The `data` property of the `AWS.Response` object contains the serialized data returned by the service request. When the request is successful, the `data` property contains an object that contains a map to the data returned. The `data` property can be null if an error occurs.

Here is an example of calling the `getItem` method of a DynamoDB table to retrieve the file name of an image file to use as part of a game.

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

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

For this example, the DynamoDB table is a lookup of images that show the results of a slot machine pull as specified by the parameters in `slotParams`.

Upon a successful call of the `getItem` method, the `data` property of the `AWS.Response` object contains an `Item` object returned by DynamoDB. The returned data is accessed according to the request's `ProjectionExpression` parameter, which in this case means the `imageFile` member of the `Item` object. Because the `imageFile` member holds a string value, you access the file name of the image itself through the value of the `S` child member of `imageFile`.

Paging Through Returned Data

Sometimes the contents of the `data` property returned by a service request span multiple pages. You can access the next page of data by calling the `response.nextPage` method. This method

sends a new request. The response from the request can be captured either with a callback or with success and error listeners.

You can check to see if the data returned by a service request has additional pages of data by calling the `response.hasNextPage` method. This method returns a boolean to indicate whether calling `response.nextPage` returns additional data.

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

Accessing Error Information from a Response Object

The `error` property of the `AWS.Response` object contains the available error data in the event of a service error or transfer error. The error returned takes the following form.

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

In the case of an error, the value of the `data` property is `null`. If you handle events that can be in a failure state, always check whether the `error` property was set before attempting to access the value of the `data` property.

Accessing the Originating Request Object

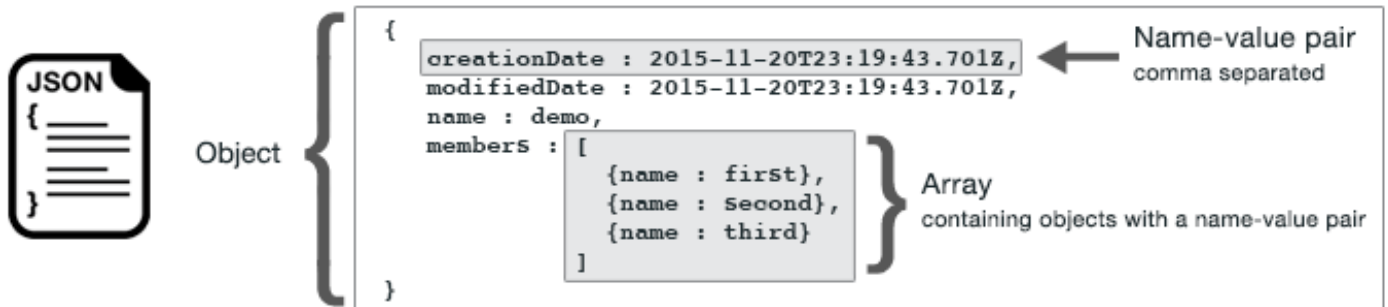
The `request` property provides access to the originating `AWS.Request` object. It can be useful to refer to the original `AWS.Request` object to access the original parameters it sent. In the following example, the `request` property is used to access the `Key` parameter of the original service request.

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

Working with JSON

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

The 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:

- 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.
- 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 a Lambda service object.

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

The `pullParams` 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.

```
lambda = new AWS.Lambda({region: 'us-west-2', apiVersion: '2015-03-31'});
// create JSON object for service call parameters
var pullParams = {
  FunctionName : 'slotPull',
  InvocationType : 'RequestResponse',
  LogType : 'None'
};
// invoke Lambda function, passing JSON object
lambda.invoke(pullParams, function(err, data) {
  if (err) {
    console.log(err);
  } else {
    console.log(data);
  }
});
```

Returning Data as JSON

JSON provides a standard way to pass data between parts of an application that need to send several values at the same time. The methods of client classes in the API commonly return JSON

in the data parameter passed to their callback functions. For example, here is a call to the `getBucketCors` method of the Amazon S3 client class.

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

The value of data is a JSON object, in this example JSON that describes the current CORS configuration for a specified Amazon S3 bucket.

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

Retry strategy in the AWS SDK for JavaScript v2

Numerous components on a network, such as DNS servers, switches, load balancers, and others can generate errors anywhere in the life of a given request. The usual technique for dealing with these error responses in a networked environment is to implement retries in the client application. This technique increases the reliability of the application and reduces operational costs for the developer. AWS SDKs implement automated retry logic for your AWS requests.

Exponential backoff based retry behavior

The AWS SDK for JavaScript v2 implements retry logic using [exponential backoff with full jitter](#) for better flow control. The idea behind exponential backoff is to use progressively longer waits

between retries for consecutive error responses. The jitter (randomized delay) is used to prevent successive collisions.

Testing retry delay in v2

In order to test retry delay in v2, the code in [node_modules/aws-sdk/lib/event_listeners.js](#) was updated to `console.log` the value present in variable `delay` as follows:

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

Retry delays with default configuration

You can test delay for any operation on AWS SDK clients. We call `listTables` operation on a DynamoDB client using the following code:

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

const region = "us-east-1";
const client = new AWS.DynamoDB({ region });
await client.listTables({}).promise();
```

In order to test retries, we simulate `NetworkingError` by disconnecting internet from the device running the test code. You can also set up proxy to return a custom Error.

On running the code, you can see that retry delay using exponential backoff with jitter as follows:

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

As retry uses jitter, you will get different values in your run of the example code.

Retry delays with custom base

The AWS SDK for JavaScript v2 allows passing a custom base number of milliseconds to use in the exponential backoff for operation retries. It defaults to 100 ms for all services except DynamoDB, where it defaults to 50 ms.

We test retries with a custom base of 1000 ms as follows:

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

We simulate `NetworkingError` by disconnecting internet from the device running the test code. You can see that the values for retry delay are higher as compared to previous run where the default was 50 or 100 ms.

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

As retry uses jitter, you will get different values in your run of the example code.

Retry delays with custom backoff algorithm

The AWS SDK for JavaScript v2 also allows passing a custom backoff function that accepts a retry count and error and returns the amount of time to delay in milliseconds. If the result is a non-zero negative value, no further retry attempts will be made.

We test custom backoff function which uses linear backoff with base value of 200 ms as follows:

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

We simulate `NetworkingError` by disconnecting internet from the device running the test code. You can see that the values for retry delay are multiples of 200.

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

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

Topics

- [Amazon CloudWatch Examples](#)
- [Amazon DynamoDB Examples](#)
- [Amazon EC2 Examples](#)
- [AWS Elemental MediaConvert Examples](#)
- [AWS IAM Examples](#)
- [Amazon Kinesis Example](#)
- [Amazon S3 Examples](#)
- [Amazon Simple Email Service Examples](#)
- [Amazon Simple Notification Service Examples](#)
- [Amazon SQS Examples](#)

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 `AWS.CloudWatch`, `AWS.CloudWatchEvents`, and `AWS.CloudWatchLogs` client classes. For more information about using the CloudWatch client classes, see [Class: AWS.CloudWatch](#), [Class: AWS.CloudWatchEvents](#), and [Class: AWS.CloudWatchLogs](#) in the API reference.

Topics

- [Creating Alarms in Amazon CloudWatch](#)
- [Using Alarm Actions in Amazon CloudWatch](#)
- [Getting Metrics from Amazon CloudWatch](#)
- [Sending Events to Amazon CloudWatch Events](#)
- [Using Subscription Filters in Amazon CloudWatch Logs](#)

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 `AWS.CloudWatch` client class:

- [describeAlarms](#)
- [putMetricAlarm](#)
- [deleteAlarms](#)

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:

- Install Node.js. For more information about installing Node.js, see the [Node.js website](#).
- 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](#).

Describing Alarms

Create a Node.js module with the file name `cw_describealarms.js`. Be sure to configure the SDK as previously shown. To access CloudWatch, create an `AWS.CloudWatch` service object. 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 `describeAlarms` method of the `AWS.CloudWatch` service object.

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

// Create CloudWatch service object
```

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

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

To run the example, type the following at the command line.

```
node cw_describealarms.js
```

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

Creating an Alarm for a CloudWatch Metric

Create a Node.js module with the file name `cw_putmetricalarm.js`. Be sure to configure the SDK as previously shown. To access CloudWatch, create an `AWS.CloudWatch` service object. 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 `describeAlarms` method of the `AWS.CloudWatch` service object.

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

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

var params = {
  AlarmName: "Web_Server_CPU_Utilization",
  ComparisonOperator: "GreaterThanThreshold",
  EvaluationPeriods: 1,
  MetricName: "CPUUtilization",
  Namespace: "AWS/EC2",
```

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

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

To run the example, type the following at the command line.

```
node cw_putmetricalarm.js
```

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

Deleting an Alarm

Create a Node.js module with the file name `cw_deletealarms.js`. Be sure to configure the SDK as previously shown. To access CloudWatch, create an `AWS.CloudWatch` service object. Create a JSON object to hold the names of the alarms you want to delete. Then call the `deleteAlarms` method of the `AWS.CloudWatch` service object.

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

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



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

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

To run the example, type the following at the command line.

```
node cw_deletealarms.js
```

This sample 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:

- [enableAlarmActions](#)
- [disableAlarmActions](#)

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:

- Install Node.js. For more information about installing Node.js, see the [Node.js website](#).
- 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](#).
- 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*.

Use the following role policy when creating the IAM role.

JSON

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

```
}
```

Configure the SDK for JavaScript by creating a global configuration object then setting the Region for your code. In this example, the Region is set to `us-west-2`.

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

Creating and Enabling Actions on an Alarm

Create a Node.js module with the file name `cw_enablealarmactions.js`. Be sure to configure the SDK as previously shown. To access CloudWatch, create an `AWS.CloudWatch` service object.

Create a JSON object to hold the parameters for creating an alarm, specifying `ActionsEnabled` as `true` and an array of ARNs for the actions the alarm will trigger. Call the `putMetricAlarm` method of the `AWS.CloudWatch` 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 `putMetricAlarm`, upon successful completion create a JSON object containing the name of the CloudWatch alarm. Call the `enableAlarmActions` method to enable the alarm action.

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

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

var params = {
  AlarmName: "Web_Server_CPU_Utilization",
  ComparisonOperator: "GreaterThanThreshold",
  EvaluationPeriods: 1,
  MetricName: "CPUUtilization",
  Namespace: "AWS/EC2",
  Period: 60,
```

```
Statistic: "Average",
Threshold: 70.0,
ActionsEnabled: true,
AlarmActions: ["ACTION_ARN"],
AlarmDescription: "Alarm when server CPU exceeds 70%",
Dimensions: [
  {
    Name: "InstanceId",
    Value: "INSTANCE_ID",
  },
],
Unit: "Percent",
};

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

To run the example, type the following at the command line.

```
node cw_enablealarmactions.js
```

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

Disabling Actions on an Alarm

Create a Node.js module with the file name `cw_disablealarmactions.js`. Be sure to configure the SDK as previously shown. To access CloudWatch, create an `AWS.CloudWatch`

service object. Create a JSON object containing the name of the CloudWatch alarm. Call the `disableAlarmActions` method to disable the actions for this alarm.

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

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

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

To run the example, type the following at the command line.

```
node cw_disablealarmactions.js
```

This sample 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:

- [listMetrics](#)
- [putMetricData](#)

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:

- Install Node.js. For more information about installing Node.js, see the [Node.js website](#).
- 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](#).

Listing Metrics

Create a Node.js module with the file name `cw_listmetrics.js`. Be sure to configure the SDK as previously shown. To access CloudWatch, create an `AWS.CloudWatch` service object. Create a JSON object containing the parameters needed to list metrics within the `AWS/Logs` namespace. Call the `listMetrics` method to list the `IncomingLogEvents` metric.

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

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

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

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

To run the example, type the following at the command line.

```
node cw_listmetrics.js
```

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

Submitting Custom Metrics

Create a Node.js module with the file name `cw_putmetricdata.js`. Be sure to configure the SDK as previously shown. To access CloudWatch, create an `AWS.CloudWatch` service object. Create a JSON object containing the parameters needed to submit a data point for the `PAGES_VISITED` custom metric. Call the `putMetricData` method.

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

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

// Create parameters JSON for putMetricData
var params = {
  MetricData: [
```

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

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

To run the example, type the following at the command line.

```
node cw_putmetricdata.js
```

This sample 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:

- [putRule](#)
- [putTargets](#)
- [putEvents](#)

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:

- Install Node.js. For more information about installing Node.js, see the [Node.js website](#).
- 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](#).
- Create a Lambda function using the **hello-world** blueprint to serve as the target for events. To learn how, see [Step 1: Create an AWS 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.

JSON

```
{
```

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

Use the following trust relationship when creating the IAM role.

JSON

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

Creating a Scheduled Rule

Create a Node.js module with the file name `cwe_putrule.js`. Be sure to configure the SDK as previously shown. To access CloudWatch Events, create an `AWS.CloudWatchEvents` 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 `putRule` method to create the rule. The callback returns the ARN of the new or updated rule.

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

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

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

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

To run the example, type the following at the command line.

```
node cwe_putrule.js
```

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

Adding a AWS Lambda Function Target

Create a Node.js module with the file name `cwe_puttargets.js`. Be sure to configure the SDK as previously shown. To access CloudWatch Events, create an `AWS.CloudWatchEvents` service object. Create a JSON object containing the parameters needed to specify the rule to which

you want to attach the target, including the ARN of the Lambda function you created. Call the `putTargets` method of the `AWS.CloudWatchEvents` service object.

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

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

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

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

To run the example, type the following at the command line.

```
node cwe_puttargets.js
```

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

Sending Events

Create a Node.js module with the file name `cwe_putevents.js`. Be sure to configure the SDK as previously shown. To access CloudWatch Events, create an `AWS.CloudWatchEvents` 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 `putEvents` method of the `AWS.CloudWatchEvents` service object.

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

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

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

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

To run the example, type the following at the command line.

```
node cwe_putevents.js
```

This sample 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:

- [putSubscriptionFilters](#)
- [describeSubscriptionFilters](#)
- [deleteSubscriptionFilter](#)

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:

- Install Node.js. For more information about installing Node.js, see the [Node.js website](#).
- 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](#).
- Create a Lambda function as the destination for log events. You will need to use the ARN of this function. For more information about setting up a Lambda function, see [Subscription Filters with AWS 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.

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents"
      ],
      "Resource": "arn:aws:logs:*:*:*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "lambda:InvokeFunction"
      ],
      "Resource": [
        "*"
      ]
    }
  ]
}
```

Describing Existing Subscription Filters

Create a Node.js module with the file name `cwl_describesubscriptionfilters.js`. Be sure to configure the SDK as previously shown. To access CloudWatch Logs, create an `AWS.CloudWatchLogs` service object. 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 you want described. Call the `describeSubscriptionFilters` method.

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

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

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

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

To run the example, type the following at the command line.

```
node cwl_describesubscriptionfilters.js
```

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

Creating a Subscription Filter

Create a Node.js module with the file name `cwl_putsubscriptionfilter.js`. Be sure to configure the SDK as previously shown. To access CloudWatch Logs, create an `AWS.CloudWatchLogs` service object. 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 `putSubscriptionFilters` method.

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

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

var params = {
```



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

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

To run the example, type the following at the command line.

```
node cwl_putsubscriptionfilter.js
```

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

Deleting a Subscription Filter

Create a Node.js module with the file name `cwl_deletesubscriptionfilters.js`. Be sure to configure the SDK as previously shown. To access CloudWatch Logs, create an `AWS.CloudWatchLogs` service object. Create a JSON object containing the parameters needed to delete a filter, including the names of the filter and the log group. Call the `deleteSubscriptionFilters` method.

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

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

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

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

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

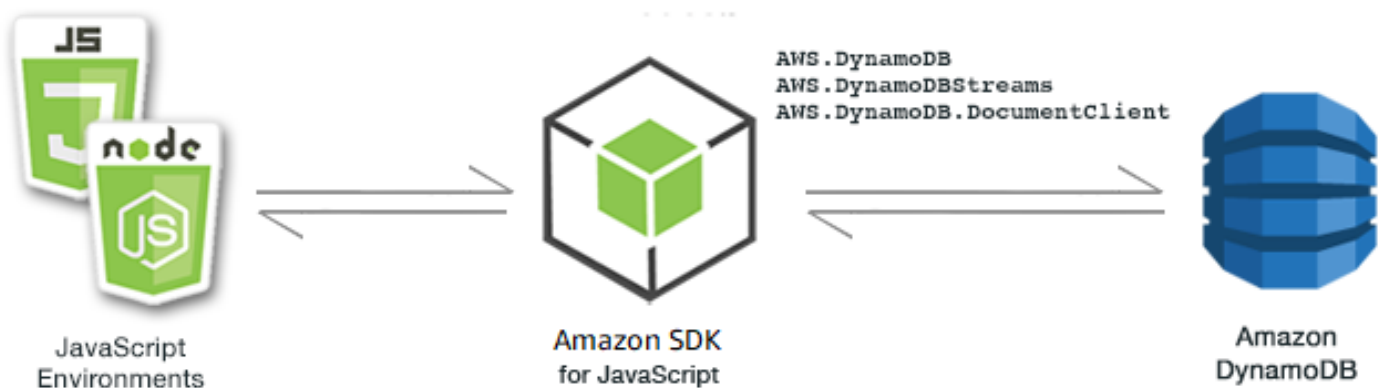
To run the example, type the following at the command line.

```
node cwl_deletesubscriptionfilter.js
```

This sample 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 `AWS.DynamoDB`, `AWS.DynamoDBStreams`, and `AWS.DynamoDB.DocumentClient` client classes. For more information about using the DynamoDB client classes, see [Class: `AWS.DynamoDB`](#), [Class: `AWS.DynamoDBStreams`](#), and [Class: `AWS.DynamoDB.DocumentClient`](#) in the API reference.

Topics

- [Creating and Using Tables in DynamoDB](#)
- [Reading and Writing A Single Item in DynamoDB](#)
- [Reading and Writing Items in Batch in DynamoDB](#)

- [Querying and Scanning a DynamoDB Table](#)
- [Using the DynamoDB Document Client](#)

Creating and Using Tables in DynamoDB



This Node.js code example shows:

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

The Scenario

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

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

- [createTable](#)
- [listTables](#)
- [describeTable](#)
- [deleteTable](#)

Prerequisite Tasks

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

- Install Node.js. For more information, see the [Node.js website](#).
- 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](#).

Creating a Table

Create a Node.js module with the file name `ddb_createtable.js`. Be sure to configure the SDK as previously shown. To access DynamoDB, create an `AWS.DynamoDB` 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 `createTable` method of the DynamoDB service object.

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

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

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

```
    StreamEnabled: false,
  },
};

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

To run the example, type the following at the command line.

```
node ddb_createtable.js
```

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

Listing Your Tables

Create a Node.js module with the file name `ddb_listtables.js`. Be sure to configure the SDK as previously shown. To access DynamoDB, create an `AWS.DynamoDB` 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 `listTables` method of the DynamoDB service object.

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

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

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

To run the example, type the following at the command line.

```
node ddb_listtables.js
```

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

Describing a Table

Create a Node.js module with the file name `ddb_describetable.js`. Be sure to configure the SDK as previously shown. To access DynamoDB, create an `AWS.DynamoDB` service object. Create a JSON object containing the parameters needed to describe a table, which in this example includes the name of the table provided as a command-line parameter. Call the `describeTable` method of the `DynamoDB` service object.

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

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

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

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

To run the example, type the following at the command line.

```
node ddb_describetable.js TABLE_NAME
```

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

Deleting a Table

Create a Node.js module with the file name `ddb_deletetable.js`. Be sure to configure the SDK as previously shown. To access DynamoDB, create an `AWS.DynamoDB` 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 `deleteTable` method of the DynamoDB service object.

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

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

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

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

To run the example, type the following at the command line.

```
node ddb_deletetable.js TABLE_NAME
```

This sample 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 `AWS.DynamoDB` client class:

- [putItem](#)
- [getItem](#)
- [deleteItem](#)

Prerequisite Tasks

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

- Install Node.js. For more information, see the [Node.js website](#).
- 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](#).
- Create a DynamoDB table whose items you can access. For more information about creating a DynamoDB table, see [Creating and Using Tables in DynamoDB](#).

Writing an Item

Create a Node.js module with the file name `ddb_putitem.js`. Be sure to configure the SDK as previously shown. To access DynamoDB, create an `AWS.DynamoDB` 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 `putItem` method of the DynamoDB service object.

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

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

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

// Call DynamoDB to add the item to the table
ddb.putItem(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

To run the example, type the following at the command line.

```
node ddb_putitem.js
```

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

Getting an Item

Create a Node.js module with the file name `ddb_getitem.js`. Be sure to configure the SDK as previously shown. To access DynamoDB, create an `AWS.DynamoDB` 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 `getItem` 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 `getItem` method of the DynamoDB service object.

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

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

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

// Call DynamoDB to read the item from the table
ddb.getItem(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.Item);
  }
});
```

To run the example, type the following at the command line.

```
node ddb_getitem.js
```

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

Deleting an Item

Create a Node.js module with the file name `ddb_deleteitem.js`. Be sure to configure the SDK as previously shown. To access DynamoDB, create an `AWS.DynamoDB` 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 `deleteItem` method of the DynamoDB service object.

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

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

var params = {
  TableName: "TABLE",
  Key: {
    KEY_NAME: { N: "VALUE" },
  },
};

// Call DynamoDB to delete the item from the table
ddb.deleteItem(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

To run the example, type the following at the command line.

```
node ddb_deleteitem.js
```

This sample 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 as well as 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:

- [batchGetItem](#)
- [batchWriteItem](#)

Prerequisite Tasks

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

- Install Node.js. For more information, see the [Node.js website](#).
- 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](#).
- Create a DynamoDB table whose items you can access. For more information about creating a DynamoDB table, see [Creating and Using Tables in DynamoDB](#).

Reading Items in Batch

Create a Node.js module with the file name `ddb_batchgetitem.js`. Be sure to configure the SDK as previously shown. To access DynamoDB, create an `AWS.DynamoDB` 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 `batchGetItem` method of the DynamoDB service object.

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

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

```
var params = {
  RequestItems: {
    TABLE_NAME: {
      Keys: [
        { KEY_NAME: { N: "KEY_VALUE_1" } },
        { KEY_NAME: { N: "KEY_VALUE_2" } },
        { KEY_NAME: { N: "KEY_VALUE_3" } },
      ],
      ProjectionExpression: "KEY_NAME, ATTRIBUTE",
    },
  },
};

ddb.batchGetItem(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    data.Responses.TABLE_NAME.forEach(function (element, index, array) {
      console.log(element);
    });
  }
});
```

To run the example, type the following at the command line.

```
node ddb_batchgetitem.js
```

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

Writing Items in Batch

Create a Node.js module with the file name `ddb_batchwriteitem.js`. Be sure to configure the SDK as previously shown. To access DynamoDB, create an `AWS.DynamoDB` 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 key(s) you want to write for each item, and the attributes along with their values. Call the `batchWriteItem` method of the DynamoDB service object.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
```

```
AWS.config.update({ region: "REGION" });

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

var params = {
  RequestItems: {
    TABLE_NAME: [
      {
        PutRequest: {
          Item: {
            KEY: { N: "KEY_VALUE" },
            ATTRIBUTE_1: { S: "ATTRIBUTE_1_VALUE" },
            ATTRIBUTE_2: { N: "ATTRIBUTE_2_VALUE" },
          },
        },
      },
      {
        PutRequest: {
          Item: {
            KEY: { N: "KEY_VALUE" },
            ATTRIBUTE_1: { S: "ATTRIBUTE_1_VALUE" },
            ATTRIBUTE_2: { N: "ATTRIBUTE_2_VALUE" },
          },
        },
      },
    ],
  },
};

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

To run the example, type the following at the command line.

```
node ddb_batchwriteitem.js
```

This sample 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:

- [query](#)
- [scan](#)

Prerequisite Tasks

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

- Install Node.js. For more information, see the [Node.js website](#).
- 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](#).
- Create a DynamoDB table whose items you can access. For more information about creating a DynamoDB table, see [Creating and Using Tables in DynamoDB](#).

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

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

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

var params = {
  ExpressionAttributeValues: {
    ":s": { N: "2" },
    ":e": { N: "09" },
    ":topic": { S: "PHRASE" },
  },
  KeyConditionExpression: "Season = :s and Episode > :e",
  ProjectionExpression: "Episode, Title, Subtitle",
  FilterExpression: "contains (Subtitle, :topic)",
  TableName: "EPISODES_TABLE",
};

ddb.query(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    //console.log("Success", data.Items);
    data.Items.forEach(function (element, index, array) {
      console.log(element.Title.S + " (" + element.Subtitle.S + ")");
    });
  }
});
```



```
}  
});
```

To run the example, type the following at the command line.

```
node ddb_query.js
```

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

Scanning a Table

Create a Node.js module with the file name `ddb_scan.js`. Be sure to configure the SDK as previously shown. To access DynamoDB, create an `AWS.DynamoDB` 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 `scan` method of the `DynamoDB` service object.

```
// Load the AWS SDK for Node.js.  
var AWS = require("aws-sdk");  
// Set the AWS Region.  
AWS.config.update({ region: "REGION" });  
  
// Create DynamoDB service object.  
var ddb = new AWS.DynamoDB({ apiVersion: "2012-08-10" });  
  
const params = {  
  // Specify which items in the results are returned.  
  FilterExpression: "Subtitle = :topic AND Season = :s AND Episode = :e",  
  // Define the expression attribute value, which are substitutes for the values you  
  // want to compare.  
  ExpressionAttributeValues: {  
    ":topic": { S: "SubTitle2" },  
    ":s": { N: 1 },  
    ":e": { N: 2 },  
  },  
  // Set the projection expression, which are the attributes that you want.  
  ProjectionExpression: "Season, Episode, Title, Subtitle",  
  TableName: "EPISODES_TABLE",  
};  
  
ddb.scan(params, function (err, data) {
```

```
if (err) {
    console.log("Error", err);
} else {
    console.log("Success", data);
    data.Items.forEach(function (element, index, array) {
        console.log(
            "printing",
            element.Title.S + " (" + element.Subtitle.S + ")"
        );
    });
}
```

To run the example, type the following at the command line.

```
node ddb_scan.js
```

This sample 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 document client.

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, as well as converts annotated response data to native JavaScript types.

For more information on the DynamoDB Document Client class, see [AWS.DynamoDB.DocumentClient](#) in the API Reference. For more information on 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 the document client. The code uses the SDK for JavaScript to query and scan tables using these methods of the DynamoDB Document Client class:

- [get](#)
- [put](#)
- [update](#)
- [query](#)
- [delete](#)

Prerequisite Tasks

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

- Install Node.js. For more information, see the [Node.js website](#).
- 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](#).
- Create a DynamoDB table whose items you can access. For more information about creating a DynamoDB table using the SDK for JavaScript, see [Creating and Using Tables in DynamoDB](#). You can also use the [DynamoDB console](#) to create a table.

Getting an Item from a Table

Create a Node.js module with the file name `ddbdoc_get.js`. Be sure to configure the SDK as previously shown. To access DynamoDB, create an `AWS.DynamoDB.DocumentClient` object. 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 `get` method of the DynamoDB document client.

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

// Create DynamoDB document client
var docClient = new AWS.DynamoDB.DocumentClient({ apiVersion: "2012-08-10" });
```

```
var params = {
  TableName: "EPISODES_TABLE",
  Key: { KEY_NAME: VALUE },
};

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

To run the example, type the following at the command line.

```
node ddbdoc_get.js
```

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

Putting an Item in a Table

Create a Node.js module with the file name `ddbdoc_put.js`. Be sure to configure the SDK as previously shown. To access DynamoDB, create an `AWS.DynamoDB.DocumentClient` object. 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 as well as names and values for attributes to set on the item. Call the `put` method of the DynamoDB document client.

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

// Create DynamoDB document client
var docClient = new AWS.DynamoDB.DocumentClient({ apiVersion: "2012-08-10" });

var params = {
  TableName: "TABLE",
  Item: {
    HASHKEY: VALUE,
    ATTRIBUTE_1: "STRING_VALUE",
```

```
    ATTRIBUTE_2: VALUE_2,
  },
};

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

To run the example, type the following at the command line.

```
node ddbdoc_put.js
```

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

Updating an Item in a Table

Create a Node.js module with the file name `ddbdoc_update.js`. Be sure to configure the SDK as previously shown. To access DynamoDB, create an `AWS.DynamoDB.DocumentClient` object. 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 `update` method of the DynamoDB document client.

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

// Create DynamoDB document client
var docClient = new AWS.DynamoDB.DocumentClient({ apiVersion: "2012-08-10" });

// Create variables to hold numeric key values
var season = SEASON_NUMBER;
var episode = EPISODES_NUMBER;

var params = {
```

```
TableName: "EPISODES_TABLE",
Key: {
  Season: season,
  Episode: episode,
},
UpdateExpression: "set Title = :t, Subtitle = :s",
ExpressionAttributeValues: {
  ":t": "NEW_TITLE",
  ":s": "NEW_SUBTITLE",
},
};

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

To run the example, type the following at the command line.

```
node ddbdoc_update.js
```

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

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 Node.js module with the file name `ddbdoc_query.js`. Be sure to configure the SDK as previously shown. To access DynamoDB, create an `AWS.DynamoDB.DocumentClient` 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, and a `KeyConditionExpression` that uses those values to define which items the query returns. Call the `query` method of the DynamoDB document client.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
```

```
// Set the region
AWS.config.update({ region: "REGION" });

// Create DynamoDB document client
var docClient = new AWS.DynamoDB.DocumentClient({ apiVersion: "2012-08-10" });

var params = {
  ExpressionAttributeValues: {
    ":s": 2,
    ":e": 9,
    ":topic": "PHRASE",
  },
  KeyConditionExpression: "Season = :s and Episode > :e",
  FilterExpression: "contains (Subtitle, :topic)",
  TableName: "EPISODES_TABLE",
};

docClient.query(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.Items);
  }
});
```

To run the example, type the following at the command line.

```
node ddbdoc_query.js
```

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

Deleting an Item from a Table

Create a Node.js module with the file name `ddbdoc_delete.js`. Be sure to configure the SDK as previously shown. To access DynamoDB, create an `AWS.DynamoDB.DocumentClient` 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 as well as a the name and value of the hashkey of the item you want to delete. Call the `delete` method of the DynamoDB document client.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
```

```
AWS.config.update({ region: "REGION" });

// Create DynamoDB document client
var docClient = new AWS.DynamoDB.DocumentClient({ apiVersion: "2012-08-10" });

var params = {
  Key: {
    HASH_KEY: VALUE,
  },
  TableName: "TABLE",
};

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

To run the example, type the following at the command line.

```
node ddbdoc_delete.js
```

This sample 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 resizable compute capacity.



The JavaScript API for Amazon EC2 is exposed through the `AWS.EC2` client class. For more information about using the Amazon EC2 client class, see [Class: `AWS.EC2`](#) in the API reference.

Topics

- [Creating an Amazon EC2 Instance](#)
- [Managing Amazon EC2 Instances](#)
- [Working with Amazon EC2 Key Pairs](#)
- [Using Regions and Availability Zones with Amazon EC2](#)
- [Working with Security Groups in Amazon EC2](#)
- [Using Elastic IP Addresses in Amazon EC2](#)

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:

- [runInstances](#)
- [createTags](#)

Prerequisite Tasks

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

- Install Node.js. For more information, see the [Node.js website](#).
- 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](#).
- Create a key pair. For details, see [Working with Amazon EC2 Key Pairs](#). You use the name of the key pair in this example.

Creating and Tagging an Instance

Create a Node.js module with the file name `ec2_createinstances.js`. Be sure to configure the SDK as previously shown.

Create an object to pass the parameters for the `runInstances` method of the `AWS.EC2` client class, including the name of the key pair to assign and the ID of the AMI to run. To call the `runInstances` method, create a promise for invoking an Amazon EC2 service object, passing the parameters. Then handle the response in the promise callback.

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 `createTags` method.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Load credentials and set region from JSON file
AWS.config.update({ region: "REGION" });

// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

// AMI is amzn-ami-2011.09.1.x86_64-ebs
var instanceParams = {
  ImageId: "AMI_ID",
  InstanceType: "t2.micro",
  KeyName: "KEY_PAIR_NAME",
  MinCount: 1,
  MaxCount: 1,
};

// Create a promise on an EC2 service object
var instancePromise = new AWS.EC2({ apiVersion: "2016-11-15" })
```

```
.runInstances(instanceParams)
.promise();

// Handle promise's fulfilled/rejected states
instancePromise
.then(function (data) {
  console.log(data);
  var instanceId = data.Instances[0].InstanceId;
  console.log("Created instance", instanceId);
  // Add tags to the instance
  tagParams = {
    Resources: [instanceId],
    Tags: [
      {
        Key: "Name",
        Value: "SDK Sample",
      },
    ],
  };
  // Create a promise on an EC2 service object
  var tagPromise = new AWS.EC2({ apiVersion: "2016-11-15" })
    .createTags(tagParams)
    .promise();
  // Handle promise's fulfilled/rejected states
  tagPromise
    .then(function (data) {
      console.log("Instance tagged");
    })
    .catch(function (err) {
      console.error(err, err.stack);
    });
})
.catch(function (err) {
  console.error(err, err.stack);
});
```

To run the example, type the following at the command line.

```
node ec2_createinstances.js
```

This sample 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:

- [describeInstances](#)
- [monitorInstances](#)
- [unmonitorInstances](#)
- [startInstances](#)
- [stopInstances](#)
- [rebootInstances](#)

For more information about the lifecycle of Amazon EC2 instances, see [Instance Lifecycle](#) in the *Amazon EC2 User Guide*.

Prerequisite Tasks

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

- Install Node.js. For more information about installing Node.js, see the [Node.js website](#).

- 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](#).
- Create an Amazon EC2 instance. For more information about creating Amazon EC2 instances, see [Amazon EC2 Instances](#) in the *Amazon EC2 User Guide* or [Amazon EC2 Instances](#) in the *Amazon EC2 User Guide*.

Describing Your Instances

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

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

// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

var params = {
  DryRun: false,
};

// Call EC2 to retrieve policy for selected bucket
ec2.describeInstances(params, function (err, data) {
  if (err) {
    console.log("Error", err.stack);
  } else {
    console.log("Success", JSON.stringify(data));
  }
});
```

To run the example, type the following at the command line.

```
node ec2_describeinstances.js
```

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

Managing Instance Monitoring

Create a Node.js module with the file name `ec2_monitorinstances.js`. Be sure to configure the SDK as previously shown. To access Amazon EC2, create an `AWS.EC2` service object. 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 `monitorInstances` method of the Amazon EC2 service object to begin detailed monitoring of the specified instances or call the `unmonitorInstances` method. Use the `DryRun` parameter to test whether you have permission to change instance monitoring before you attempt to change the monitoring of these instances.

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

// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

var params = {
  InstanceIds: ["INSTANCE_ID"],
  DryRun: true,
};

if (process.argv[2].toUpperCase() === "ON") {
  // Call EC2 to start monitoring the selected instances
  ec2.monitorInstances(params, function (err, data) {
    if (err && err.code === "DryRunOperation") {
      params.DryRun = false;
      ec2.monitorInstances(params, function (err, data) {
        if (err) {
          console.log("Error", err);
        } else if (data) {
          console.log("Success", data.InstanceMonitorings);
        }
      });
    } else {
      console.log("You don't have permission to change instance monitoring.");
    }
  });
} else if (process.argv[2].toUpperCase() === "OFF") {
```

```
// Call EC2 to stop monitoring the selected instances
ec2.unmonitorInstances(params, function (err, data) {
  if (err && err.code === "DryRunOperation") {
    params.DryRun = false;
    ec2.unmonitorInstances(params, function (err, data) {
      if (err) {
        console.log("Error", err);
      } else if (data) {
        console.log("Success", data.InstanceMonitorings);
      }
    });
  } else {
    console.log("You don't have permission to change instance monitoring.");
  }
});
}
```

To run the example, type the following at the command line, specifying ON to begin detailed monitoring or OFF to discontinue monitoring.

```
node ec2_monitorinstances.js ON
```

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

Starting and Stopping Instances

Create a Node.js module with the file name `ec2_startstopinstances.js`. Be sure to configure the SDK as previously shown. To access Amazon EC2, create an `AWS.EC2` service object. 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 `startInstances` method of the Amazon EC2 service object to start the specified instances, or the `stopInstances` method to stop them. Use the `DryRun` parameter to test whether you have permission before actually attempting to start or stop the selected instances.

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

// Create EC2 service object
```

```
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

var params = {
  InstanceIds: [process.argv[3]],
  DryRun: true,
};

if (process.argv[2].toUpperCase() === "START") {
  // Call EC2 to start the selected instances
  ec2.startInstances(params, function (err, data) {
    if (err && err.code === "DryRunOperation") {
      params.DryRun = false;
      ec2.startInstances(params, function (err, data) {
        if (err) {
          console.log("Error", err);
        } else if (data) {
          console.log("Success", data.StartingInstances);
        }
      });
    } else {
      console.log("You don't have permission to start instances.");
    }
  });
} else if (process.argv[2].toUpperCase() === "STOP") {
  // Call EC2 to stop the selected instances
  ec2.stopInstances(params, function (err, data) {
    if (err && err.code === "DryRunOperation") {
      params.DryRun = false;
      ec2.stopInstances(params, function (err, data) {
        if (err) {
          console.log("Error", err);
        } else if (data) {
          console.log("Success", data.StoppingInstances);
        }
      });
    } else {
      console.log("You don't have permission to stop instances");
    }
  });
}
```

To run the example, type the following at the command line specifying START to start the instances or STOP to stop them.


```
node ec2_startstopinstances.js START INSTANCE_ID
```

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

Rebooting Instances

Create a Node.js module with the file name `ec2_rebootinstances.js`. Be sure to configure the SDK as previously shown. To access Amazon EC2, create an Amazon EC2 service object. Add the instance IDs of the instances you want to reboot. Call the `rebootInstances` method of the `AWS.EC2` service object to reboot the specified instances. Use the `DryRun` parameter to test whether you have permission to reboot these instances before actually attempting to reboot them.

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

// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

var params = {
  InstanceIds: ["INSTANCE_ID"],
  DryRun: true,
};

// Call EC2 to reboot instances
ec2.rebootInstances(params, function (err, data) {
  if (err && err.code === "DryRunOperation") {
    params.DryRun = false;
    ec2.rebootInstances(params, function (err, data) {
      if (err) {
        console.log("Error", err);
      } else if (data) {
        console.log("Success", data);
      }
    });
  } else {
    console.log("You don't have permission to reboot instances.");
  }
});
```

To run the example, type the following at the command line.

```
node ec2_rebootinstances.js
```

This sample 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:

- [createKeyPair](#)
- [deleteKeyPair](#)
- [describeKeyPairs](#)

For more information about the Amazon EC2 key pairs, see [Amazon EC2 Key Pairs](#) in the *Amazon EC2 User Guide* or [Amazon EC2 Key Pairs and Windows Instances](#) in the *Amazon EC2 User Guide*.

Prerequisite Tasks

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

- Install Node.js. For more information about installing Node.js, see the [Node.js website](#).
- 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](#).

Describing Your Key Pairs

Create a Node.js module with the file name `ec2_describekeypairs.js`. Be sure to configure the SDK as previously shown. To access Amazon EC2, create an `AWS.EC2` service object. Create an empty JSON object to hold the parameters needed by the `describeKeyPairs` 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 `describeKeyPairs` method.

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

// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

// Retrieve key pair descriptions; no params needed
ec2.describeKeyPairs(function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", JSON.stringify(data.KeyPairs));
  }
});
```

To run the example, type the following at the command line.

```
node ec2_describekeypairs.js
```

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

Creating a Key Pair

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. To access Amazon EC2, create an `AWS.EC2` service object. Create the JSON parameters to specify the name of the key pair, then pass them to call the `createKeyPair` method.

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

// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

var params = {
  KeyName: "KEY_PAIR_NAME",
};

// Create the key pair
ec2.createKeyPair(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log(JSON.stringify(data));
  }
});
```

To run the example, type the following at the command line.

```
node ec2_createkeypair.js
```

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

Deleting a Key Pair

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

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
```

```
AWS.config.update({ region: "REGION" });

// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

var params = {
  KeyName: "KEY_PAIR_NAME",
};

// Delete the key pair
ec2.deleteKeyPair(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Key Pair Deleted");
  }
});
```

To run the example, type the following at the command line.

```
node ec2_deletekeypair.js
```

This sample 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 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:

- [describeAvailabilityZones](#)
- [describeRegions](#)

For more information about Regions and Availability Zones, see [Regions and Availability Zones](#) in the *Amazon EC2 User Guide* or [Regions and Availability Zones](#) in the *Amazon EC2 User Guide*.

Prerequisite Tasks

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

- Install Node.js. For more information about installing Node.js, see the [Node.js website](#).
- 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](#).

Describing Regions and Availability Zones

Create a Node.js module with the file name `ec2_describeregionsandzones.js`. Be sure to configure the SDK as previously shown. To access Amazon EC2, create an `AWS.EC2` service object. Create an empty JSON object to pass as parameters, which returns all available descriptions. Then call the `describeRegions` and `describeAvailabilityZones` methods.

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

// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

var params = {};

// Retrieves all regions/endpoints that work with EC2
ec2.describeRegions(params, function (err, data) {
  if (err) {
```

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

// Retrieves availability zones only for region of the ec2 service object
ec2.describeAvailabilityZones(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Availability Zones: ", data.AvailabilityZones);
  }
});
```

To run the example, type the following at the command line.

```
node ec2_describeregionsandzones.js
```

This sample 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:

- [describeSecurityGroups](#)
- [authorizeSecurityGroupIngress](#)
- [createSecurityGroup](#)
- [describeVpcs](#)
- [deleteSecurityGroup](#)

For more information about the Amazon EC2 security groups, see [Amazon EC2 Amazon Security Groups for Linux Instances](#) in the *Amazon EC2 User Guide* or [Amazon EC2 Security Groups for Windows Instances](#) in the *Amazon EC2 User Guide*.

Prerequisite Tasks

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

- Install Node.js. For more information about installing Node.js, see the [Node.js website](#).
- 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](#).

Describing Your Security Groups

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

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

// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });
```



```
var params = {
  GroupIds: ["SECURITY_GROUP_ID"],
};

// Retrieve security group descriptions
ec2.describeSecurityGroups(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", JSON.stringify(data.SecurityGroups));
  }
});
```

To run the example, type the following at the command line.

```
node ec2_describesecuritygroups.js
```

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

Creating a Security Group and Rules

Create a Node.js module with the file name `ec2_createsecuritygroup.js`. Be sure to configure the SDK as previously shown. To access Amazon EC2, create an `AWS.EC2` service object. 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 `createSecurityGroup` 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 `authorizeSecurityGroupIngress` method.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Load credentials and set region from JSON file
AWS.config.update({ region: "REGION" });

// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

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

```
// Retrieve the ID of a VPC
ec2.describeVpcs(function (err, data) {
  if (err) {
    console.log("Cannot retrieve a VPC", err);
  } else {
    vpc = data.Vpcs[0].VpcId;
    var paramsSecurityGroup = {
      Description: "DESCRIPTION",
      GroupName: "SECURITY_GROUP_NAME",
      VpcId: vpc,
    };
    // Create the instance
    ec2.createSecurityGroup(paramsSecurityGroup, function (err, data) {
      if (err) {
        console.log("Error", err);
      } else {
        var SecurityGroupId = data.GroupId;
        console.log("Success", SecurityGroupId);
        var paramsIngress = {
          GroupId: "SECURITY_GROUP_ID",
          IpPermissions: [
            {
              IpProtocol: "tcp",
              FromPort: 80,
              ToPort: 80,
              IpRanges: [{ CidrIp: "0.0.0.0/0" }],
            },
            {
              IpProtocol: "tcp",
              FromPort: 22,
              ToPort: 22,
              IpRanges: [{ CidrIp: "0.0.0.0/0" }],
            },
          ],
        };
        ec2.authorizeSecurityGroupIngress(paramsIngress, function (err, data) {
          if (err) {
            console.log("Error", err);
          } else {
            console.log("Ingress Successfully Set", data);
          }
        });
      }
    });
  }
}
```

```
    });  
  }  
});
```

To run the example, type the following at the command line.

```
node ec2_createsecuritygroup.js
```

This sample 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 `AWS.EC2` service object. Create the JSON parameters to specify the name of the security group to delete. Then call the `deleteSecurityGroup` method.

```
// Load the AWS SDK for Node.js  
var AWS = require("aws-sdk");  
// Set the region  
AWS.config.update({ region: "REGION" });  
  
// Create EC2 service object  
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });  
  
var params = {  
  GroupId: "SECURITY_GROUP_ID",  
};  
  
// Delete the security group  
ec2.deleteSecurityGroup(params, function (err, data) {  
  if (err) {  
    console.log("Error", err);  
  } else {  
    console.log("Security Group Deleted");  
  }  
});
```

To run the example, type the following at the command line.

```
node ec2_deletesecuritygroup.js
```

This sample 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:

- [`describeAddresses`](#)
- [`allocateAddress`](#)
- [`associateAddress`](#)
- [`releaseAddress`](#)

For more information about Elastic IP addresses in Amazon EC2, see [Elastic IP Addresses](#) in the *Amazon EC2 User Guide* or [Elastic IP Addresses](#) in the *Amazon EC2 User Guide*.

Prerequisite Tasks

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

- Install Node.js. For more information about installing Node.js, see the [Node.js website](#).

- 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](#).
- Create an Amazon EC2 instance. For more information about creating Amazon EC2 instances, see [Amazon EC2 Instances](#) in the *Amazon EC2 User Guide* or [Amazon EC2 Instances](#) in the *Amazon EC2 User Guide*.

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. To access Amazon EC2, create an `AWS.EC2` service object. 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 `describeAddresses` method of the Amazon EC2 service object.

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

// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

var params = {
  Filters: [{ Name: "domain", Values: ["vpc"] }],
};

// Retrieve Elastic IP address descriptions
ec2.describeAddresses(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", JSON.stringify(data.Addresses));
  }
});
```

To run the example, type the following at the command line.

```
node ec2_describeaddresses.js
```

This sample 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 `AWS.EC2` 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 `allocateAddress` 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 `associateAddresses` method of the Amazon EC2 service object.

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

// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

var paramsAllocateAddress = {
  Domain: "vpc",
};

// Allocate the Elastic IP address
ec2.allocateAddress(paramsAllocateAddress, function (err, data) {
  if (err) {
    console.log("Address Not Allocated", err);
  } else {
    console.log("Address allocated:", data.AllocationId);
    var paramsAssociateAddress = {
      AllocationId: data.AllocationId,
      InstanceId: "INSTANCE_ID",
    };
    // Associate the new Elastic IP address with an EC2 instance
    ec2.associateAddresses(paramsAssociateAddress, function (err, data) {
      if (err) {
```

```
        console.log("Address Not Associated", err);
    } else {
        console.log("Address associated:", data.AssociationId);
    }
});
}
```

To run the example, type the following at the command line.

```
node ec2_allocateaddress.js
```

This sample 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. To access Amazon EC2, create an `AWS.EC2` 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 `releaseAddress` method of the Amazon EC2 service object.

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

// Create EC2 service object
var ec2 = new AWS.EC2({ apiVersion: "2016-11-15" });

var paramsReleaseAddress = {
    AllocationId: "ALLOCATION_ID",
};

// Disassociate the Elastic IP address from EC2 instance
ec2.releaseAddress(paramsReleaseAddress, function (err, data) {
    if (err) {
        console.log("Error", err);
    } else {
        console.log("Address released");
    }
});
```

To run the example, type the following at the command line.

```
node ec2_releaseaddress.js
```

This sample 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 `AWS.MediaConvert` client class. For more information, see [Class: `AWS.MediaConvert`](#) in the API reference.

Topics

- [Getting Your Region-Specific Endpoint for MediaConvert](#)
- [Creating and Managing Transcoding Jobs in MediaConvert](#)
- [Using Job Templates in MediaConvert](#)

Getting Your Region-Specific Endpoint for MediaConvert



This Node.js code example shows:

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

The Scenario

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

- [describeEndpoints](#)

Important

The default Node.js HTTP/HTTPS agent creates a new TCP connection for every new request. To avoid the cost of establishing a new connection, the AWS SDK for JavaScript reuses TCP connections. For more information, see [Reusing Connections with Keep-Alive in Node.js](#).

Prerequisite Tasks

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

- Install Node.js. For more information, see the [Node.js website](#).
- 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](#).
- 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*.

Getting Your Endpoint URL

Create a Node.js module with the file name `emc_getendpoint.js`. Be sure to configure the SDK as previously shown.

Create an object to pass the empty request parameters for the `describeEndpoints` method of the `AWS.MediaConvert` client class. To call the `describeEndpoints` method, create a promise for invoking an `MediaConvert` service object, passing the parameters. Handle the response in the promise callback.

```
// Load the SDK for JavaScript.
const aws = require("aws-sdk");

// Set the AWS Region.
aws.config.update({ region: "us-west-2" });
```

```
// Create the client.
const mediaConvert = new aws.MediaConvert({ apiVersion: "2017-08-29" });

exports.handler = async (event, context) => {
  // Create empty request parameters
  const params = {
    MaxResults: 0,
  };

  try {
    const { Endpoints } = await mediaConvert
      .describeEndpoints(params)
      .promise();
    console.log("Your MediaConvert endpoint is ", Endpoints);
  } catch (err) {
    console.log("MediaConvert Error", err);
  }
};
```

To run the example, type the following at the command line.

```
node emc_getendpoint.js
```

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

Creating and Managing Transcoding Jobs in MediaConvert



This Node.js code example shows:

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

The Scenario

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

- [createJob](#)
- [cancelJob](#)
- [getJob](#)
- [listJobs](#)

Prerequisite Tasks

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

- Install Node.js. For more information, see the [Node.js website](#).
- 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](#).
- 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*.

Configuring the SDK

Configure the SDK for JavaScript by creating a global configuration object, and then setting the Region for your code. In this example, the Region is set to us-west-2. Because MediaConvert uses custom endpoints for each account, you must also configure the `AWS.MediaConvert` client class to use your region-specific endpoint. To do this, set the endpoint parameter on `AWS.config.mediaconvert`.

```
// Load the SDK for JavaScript
var AWS = require("aws-sdk");
// Set the Region
AWS.config.update({ region: "us-west-2" });
// Set the custom endpoint for your account
AWS.config.mediaconvert = { endpoint: "ACCOUNT_ENDPOINT" };
```

Defining a Simple Transcoding Job

Create a Node.js module with the file name `emc_createjob.js`. Be sure to configure the SDK as previously shown. 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.

```
var params = {
  Queue: "JOB_QUEUE_ARN",
  UserMetadata: {
    Customer: "Amazon",
  },
  Role: "IAM_ROLE_ARN",
  Settings: {
    OutputGroups: [
      {
        Name: "File Group",
        OutputGroupSettings: {
          Type: "FILE_GROUP_SETTINGS",
          FileGroupSettings: {
            Destination: "s3://OUTPUT_BUCKET_NAME/",
          },
        },
      },
    ],
    Outputs: [
      {
        VideoDescription: {
          ScalingBehavior: "DEFAULT",
          TimecodeInsertion: "DISABLED",
          AntiAlias: "ENABLED",
          Sharpness: 50,
          CodecSettings: {
            Codec: "H_264",
            H264Settings: {
```

```

        InterlaceMode: "PROGRESSIVE",
        NumberReferenceFrames: 3,
        Syntax: "DEFAULT",
        Softness: 0,
        GopClosedCadence: 1,
        GopSize: 90,
        Slices: 1,
        GopBReference: "DISABLED",
        SlowPal: "DISABLED",
        SpatialAdaptiveQuantization: "ENABLED",
        TemporalAdaptiveQuantization: "ENABLED",
        FlickerAdaptiveQuantization: "DISABLED",
        EntropyEncoding: "CABAC",
        Bitrate: 5000000,
        FramerateControl: "SPECIFIED",
        RateControlMode: "CBR",
        CodecProfile: "MAIN",
        Telecine: "NONE",
        MinIInterval: 0,
        AdaptiveQuantization: "HIGH",
        CodecLevel: "AUTO",
        FieldEncoding: "PAFF",
        SceneChangeDetect: "ENABLED",
        QualityTuningLevel: "SINGLE_PASS",
        FramerateConversionAlgorithm: "DUPLICATE_DROP",
        UnregisteredSeiTimecode: "DISABLED",
        GopSizeUnits: "FRAMES",
        ParControl: "SPECIFIED",
        NumberBFramesBetweenReferenceFrames: 2,
        RepeatPps: "DISABLED",
        FramerateNumerator: 30,
        FramerateDenominator: 1,
        ParNumerator: 1,
        ParDenominator: 1,
    },
},
AfdSignaling: "NONE",
DropFrameTimecode: "ENABLED",
RespondToAfd: "NONE",
ColorMetadata: "INSERT",
},
AudioDescriptions: [
    {
        AudioTypeControl: "FOLLOW_INPUT",

```

```

        CodecSettings: {
          Codec: "AAC",
          AacSettings: {
            AudioDescriptionBroadcasterMix: "NORMAL",
            RateControlMode: "CBR",
            CodecProfile: "LC",
            CodingMode: "CODING_MODE_2_0",
            RawFormat: "NONE",
            SampleRate: 48000,
            Specification: "MPEG4",
            Bitrate: 64000,
          },
        },
        LanguageCodeControl: "FOLLOW_INPUT",
        AudioSourceName: "Audio Selector 1",
      ],
      ContainerSettings: {
        Container: "MP4",
        Mp4Settings: {
          CslgAtom: "INCLUDE",
          FreeSpaceBox: "EXCLUDE",
          MoovPlacement: "PROGRESSIVE_DOWNLOAD",
        },
      },
      NameModifier: "_1",
    },
  ],
},
AdAvailOffset: 0,
Inputs: [
  {
    AudioSelectors: {
      "Audio Selector 1": {
        Offset: 0,
        DefaultSelection: "NOT_DEFAULT",
        ProgramSelection: 1,
        SelectorType: "TRACK",
        Tracks: [1],
      },
    },
    VideoSelector: {
      ColorSpace: "FOLLOW",

```

```
    },
    FilterEnable: "AUTO",
    PsiControl: "USE_PSI",
    FilterStrength: 0,
    DeblockFilter: "DISABLED",
    DenoiseFilter: "DISABLED",
    TimecodeSource: "EMBEDDED",
    FileInput: "s3://INPUT_BUCKET_AND_FILE_NAME",
  },
],
TimecodeConfig: {
  Source: "EMBEDDED",
},
},
};
```

Creating a Transcoding Job

After creating the job parameters JSON, call the `createJob` method by creating a promise for invoking an `AWS.MediaConvert` service object, passing the parameters. Then handle the response in the promise callback. The ID of the job created is returned in the response data.

```
// Create a promise on a MediaConvert object
var endpointPromise = new AWS.MediaConvert({ apiVersion: "2017-08-29" })
  .createJob(params)
  .promise();

// Handle promise's fulfilled/rejected status
endpointPromise.then(
  function (data) {
    console.log("Job created! ", data);
  },
  function (err) {
    console.log("Error", err);
  }
);
```

To run the example, type the following at the command line.

```
node emc_createjob.js
```

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

Canceling a Transcoding Job

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

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the Region
AWS.config.update({ region: "us-west-2" });
// Set MediaConvert to customer endpoint
AWS.config.mediaconvert = { endpoint: "ACCOUNT_ENDPOINT" };

var params = {
  Id: "JOB_ID" /* required */,
};

// Create a promise on a MediaConvert object
var endpointPromise = new AWS.MediaConvert({ apiVersion: "2017-08-29" })
  .cancelJob(params)
  .promise();

// Handle promise's fulfilled/rejected status
endpointPromise.then(
  function (data) {
    console.log("Job " + params.Id + " is canceled");
  },
  function (err) {
    console.log("Error", err);
  }
);
```

To run the example, type the following at the command line.

```
node ec2_canceljob.js
```

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

Listing Recent Transcoding Jobs

Create a Node.js module with the file name `emc_listjobs.js`. Be sure to configure the SDK as previously shown.

Create the parameters JSON, including values to specify whether to sort the list in ASCENDING, or DESCENDING order, the ARN of the job queue to check, and the status of jobs to include. Then call the `listJobs` method by creating a promise for invoking an `AWS.MediaConvert` service object, passing the parameters. Handle the response in the promise callback.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the Region
AWS.config.update({ region: "us-west-2" });
// Set the customer endpoint
AWS.config.mediaconvert = { endpoint: "ACCOUNT_ENDPOINT" };

var params = {
  MaxResults: 10,
  Order: "ASCENDING",
  Queue: "QUEUE_ARN",
  Status: "SUBMITTED",
};

// Create a promise on a MediaConvert object
var endpointPromise = new AWS.MediaConvert({ apiVersion: "2017-08-29" })
  .listJobs(params)
  .promise();

// Handle promise's fulfilled/rejected status
endpointPromise.then(
  function (data) {
    console.log("Jobs: ", data);
  },
  function (err) {
    console.log("Error", err);
  }
);
```

To run the example, type the following at the command line.

```
node emc_listjobs.js
```

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

Using Job Templates in MediaConvert



This Node.js code example shows:

- How to create 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:

- [createJobTemplate](#)
- [createJob](#)
- [deleteJobTemplate](#)
- [listJobTemplates](#)

Prerequisite Tasks

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

- Install Node.js. For more information, see the [Node.js website](#).
- 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](#).

- 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*.

Creating a Job Template

Create a Node.js module with the file name `emc_create_jobtemplate.js`. Be sure to configure the SDK as previously shown.

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

Call the `createJobTemplate` method by creating a promise for invoking an `AWS.MediaConvert` service object, passing the parameters. Then handle the response in the promise callback.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the Region
AWS.config.update({ region: "us-west-2" });
// Set the custom endpoint for your account
AWS.config.mediaconvert = { endpoint: "ACCOUNT_ENDPOINT" };

var params = {
  Category: "YouTube Jobs",
  Description: "Final production transcode",
  Name: "DemoTemplate",
  Queue: "JOB_QUEUE_ARN",
  Settings: {
    OutputGroups: [
      {
        Name: "File Group",
        OutputGroupSettings: {
          Type: "FILE_GROUP_SETTINGS",
          FileGroupSettings: {
            Destination: "s3://BUCKET_NAME/",
          },
        },
      },
    ],
    Outputs: [
      {
        VideoDescription: {
          ScalingBehavior: "DEFAULT",
        },
      },
    ],
  },
}
```

```

    TimecodeInsertion: "DISABLED",
    AntiAlias: "ENABLED",
    Sharpness: 50,
    CodecSettings: {
      Codec: "H_264",
      H264Settings: {
        InterlaceMode: "PROGRESSIVE",
        NumberReferenceFrames: 3,
        Syntax: "DEFAULT",
        Softness: 0,
        GopClosedCadence: 1,
        GopSize: 90,
        Slices: 1,
        GopBReference: "DISABLED",
        SlowPal: "DISABLED",
        SpatialAdaptiveQuantization: "ENABLED",
        TemporalAdaptiveQuantization: "ENABLED",
        FlickerAdaptiveQuantization: "DISABLED",
        EntropyEncoding: "CABAC",
        Bitrate: 5000000,
        FramerateControl: "SPECIFIED",
        RateControlMode: "CBR",
        CodecProfile: "MAIN",
        Telecine: "NONE",
        MinIInterval: 0,
        AdaptiveQuantization: "HIGH",
        CodecLevel: "AUTO",
        FieldEncoding: "PAFF",
        SceneChangeDetect: "ENABLED",
        QualityTuningLevel: "SINGLE_PASS",
        FramerateConversionAlgorithm: "DUPLICATE_DROP",
        UnregisteredSeiTimecode: "DISABLED",
        GopSizeUnits: "FRAMES",
        ParControl: "SPECIFIED",
        NumberBFramesBetweenReferenceFrames: 2,
        RepeatPps: "DISABLED",
        FramerateNumerator: 30,
        FramerateDenominator: 1,
        ParNumerator: 1,
        ParDenominator: 1,
      },
    },
    AfdSignaling: "NONE",
    DropFrameTimecode: "ENABLED",

```

```

        RespondToAfd: "NONE",
        ColorMetadata: "INSERT",
    },
    AudioDescriptions: [
        {
            AudioTypeControl: "FOLLOW_INPUT",
            CodecSettings: {
                Codec: "AAC",
                AacSettings: {
                    AudioDescriptionBroadcasterMix: "NORMAL",
                    RateControlMode: "CBR",
                    CodecProfile: "LC",
                    CodingMode: "CODING_MODE_2_0",
                    RawFormat: "NONE",
                    SampleRate: 48000,
                    Specification: "MPEG4",
                    Bitrate: 64000,
                },
            },
            LanguageCodeControl: "FOLLOW_INPUT",
            AudioSourceName: "Audio Selector 1",
        },
    ],
    ContainerSettings: {
        Container: "MP4",
        Mp4Settings: {
            CslgAtom: "INCLUDE",
            FreeSpaceBox: "EXCLUDE",
            MoovPlacement: "PROGRESSIVE_DOWNLOAD",
        },
    },
    NameModifier: "_1",
},
],
},
],
AdAvailOffset: 0,
Inputs: [
    {
        AudioSelectors: {
            "Audio Selector 1": {
                Offset: 0,
                DefaultSelection: "NOT_DEFAULT",
                ProgramSelection: 1,
            }
        }
    }
]

```

```
        SelectorType: "TRACK",
        Tracks: [1],
      },
    ],
    VideoSelector: {
      ColorSpace: "FOLLOW",
    },
    FilterEnable: "AUTO",
    PsiControl: "USE_PSI",
    FilterStrength: 0,
    DeblockFilter: "DISABLED",
    DenoiseFilter: "DISABLED",
    TimecodeSource: "EMBEDDED",
  },
],
TimecodeConfig: {
  Source: "EMBEDDED",
},
},
};

// Create a promise on a MediaConvert object
var templatePromise = new AWS.MediaConvert({ apiVersion: "2017-08-29" })
  .createJobTemplate(params)
  .promise();

// Handle promise's fulfilled/rejected status
templatePromise.then(
  function (data) {
    console.log("Success!", data);
  },
  function (err) {
    console.log("Error", err);
  }
);
```

To run the example, type the following at the command line.

```
node emc_create_jobtemplate.js
```

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

Creating a Transcoding Job from a Job Template

Create a Node.js module with the file name `emc_template_createjob.js`. Be sure to configure the SDK as previously shown.

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 `createJobs` method by creating a promise for invoking an `AWS.MediaConvert` service object, passing the parameters. Handle the response in the promise callback.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the Region
AWS.config.update({ region: "us-west-2" });
// Set the custom endpoint for your account
AWS.config.mediaconvert = { endpoint: "ACCOUNT_ENDPOINT" };

var params = {
  Queue: "QUEUE_ARN",
  JobTemplate: "TEMPLATE_NAME",
  Role: "ROLE_ARN",
  Settings: {
    Inputs: [
      {
        AudioSelectors: {
          "Audio Selector 1": {
            Offset: 0,
            DefaultSelection: "NOT_DEFAULT",
            ProgramSelection: 1,
            SelectorType: "TRACK",
            Tracks: [1],
          },
        },
        VideoSelector: {
          ColorSpace: "FOLLOW",
        },
        FilterEnable: "AUTO",
        PsiControl: "USE_PSI",
        FilterStrength: 0,
        DeblockFilter: "DISABLED",
        DenoiseFilter: "DISABLED",
        TimecodeSource: "EMBEDDED",
        FileInput: "s3://BUCKET_NAME/FILE_NAME",
```

```
    },  
    ],  
  },  
};  
  
// Create a promise on a MediaConvert object  
var templateJobPromise = new AWS.MediaConvert({ apiVersion: "2017-08-29" })  
    .createJob(params)  
    .promise();  
  
// Handle promise's fulfilled/rejected status  
templateJobPromise.then(  
    function (data) {  
        console.log("Success! ", data);  
    },  
    function (err) {  
        console.log("Error", err);  
    }  
);
```

To run the example, type the following at the command line.

```
node emc_template_createjob.js
```

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

Listing Your Job Templates

Create a Node.js module with the file name `emc_listtemplates.js`. Be sure to configure the SDK as previously shown.

Create an object to pass the request parameters for the `listTemplates` method of the `AWS.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 `listTemplates` method, create a promise for invoking an `MediaConvert` service object, passing the parameters. Then handle the response in the promise callback.

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



```
// Set the customer endpoint
AWS.config.mediaconvert = { endpoint: "ACCOUNT_ENDPOINT" };

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

// Create a promise on a MediaConvert object
var listTemplatesPromise = new AWS.MediaConvert({ apiVersion: "2017-08-29" })
  .listJobTemplates(params)
  .promise();

// Handle promise's fulfilled/rejected status
listTemplatesPromise.then(
  function (data) {
    console.log("Success ", data);
  },
  function (err) {
    console.log("Error", err);
  }
);
```

To run the example, type the following at the command line.

```
node emc_listtemplates.js
```

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

Deleting a Job Template

Create a Node.js module with the file name `emc_deletetemplate.js`. Be sure to configure the SDK as previously shown.

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

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

```
var AWS = require("aws-sdk");
// Set the Region
AWS.config.update({ region: "us-west-2" });
// Set the customer endpoint
AWS.config.mediaconvert = { endpoint: "ACCOUNT_ENDPOINT" };

var params = {
  Name: "TEMPLATE_NAME",
};

// Create a promise on a MediaConvert object
var deleteTemplatePromise = new AWS.MediaConvert({ apiVersion: "2017-08-29" })
  .deleteJobTemplate(params)
  .promise();

// Handle promise's fulfilled/rejected status
deleteTemplatePromise.then(
  function (data) {
    console.log("Success ", data);
  },
  function (err) {
    console.log("Error", err);
  }
);
```

To run the example, type the following at the command line.

```
node emc_deletetemplate.js
```

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

AWS IAM Examples

AWS Identity and Access Management (IAM) is a web service that enables Amazon Web Services 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 `AWS.IAM` client class. For more information about using the IAM client class, see [Class: `AWS.IAM`](#) in the API reference.

Topics

- [Managing IAM Users](#)
- [Working with IAM Policies](#)
- [Managing IAM Access Keys](#)
- [Working with IAM Server Certificates](#)
- [Managing IAM Account Aliases](#)

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 `AWS.IAM` client class:

- [createUser](#)
- [listUsers](#)
- [updateUser](#)
- [getUser](#)
- [deleteUser](#)

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:

- Install Node.js. For more information about installing Node.js, see the [Node.js website](#).
- 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](#).

Creating a User

Create a Node.js module with the file name `iam_createuser.js`. Be sure to configure the SDK as previously shown. To access IAM, create an `AWS.IAM` service object. 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 `getUser` method of the `AWS.IAM` service object to see if the user name already exists. If the user name does not currently exist, call the `createUser` method to create it. If the name already exists, write a message to that effect to the console.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
```

```
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

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

iam.getUser(params, function (err, data) {
  if (err && err.code === "NoSuchEntity") {
    iam.createUser(params, function (err, data) {
      if (err) {
        console.log("Error", err);
      } else {
        console.log("Success", data);
      }
    });
  } else {
    console.log(
      "User " + process.argv[2] + " already exists",
      data.User.UserId
    );
  }
});
```

To run the example, type the following at the command line.

```
node iam_createuser.js USER_NAME
```

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

Listing Users in Your Account

Create a Node.js module with the file name `iam_listusers.js`. Be sure to configure the SDK as previously shown. To access IAM, create an `AWS.IAM` service object. 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 `listUsers` method of the `AWS.IAM` service object. Write the first user's name and creation date to the console.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
```

```
// Set the region
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

var params = {
  MaxItems: 10,
};

iam.listUsers(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    var users = data.Users || [];
    users.forEach(function (user) {
      console.log("User " + user.UserName + " created", user.CreateDate);
    });
  }
});
```

To run the example, type the following at the command line.

```
node iam_listusers.js
```

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

Updating a User's Name

Create a Node.js module with the file name `iam_updateuser.js`. Be sure to configure the SDK as previously shown. To access IAM, create an `AWS.IAM` service object. 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 `updateUser` method of the `AWS.IAM` service object.

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

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });
```

```
var params = {
  UserName: process.argv[2],
  NewUserName: process.argv[3],
};

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

To run the example, type the following at the command line, specifying the user's current name followed by the new user name.

```
node iam_updateuser.js ORIGINAL_USERNAME NEW_USERNAME
```

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

Deleting a User

Create a Node.js module with the file name `iam_deleteuser.js`. Be sure to configure the SDK as previously shown. To access IAM, create an `AWS.IAM` service object. Create a JSON object containing the parameters needed, which consists of the user name you want to delete as a command-line parameter.

Call the `getUser` method of the `AWS.IAM` 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 `deleteUser` method to delete it.

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

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

var params = {
  UserName: process.argv[2],
```

```
};

iam.getUser(params, function (err, data) {
  if (err && err.code === "NoSuchEntity") {
    console.log("User " + process.argv[2] + " does not exist.");
  } else {
    iam.deleteUser(params, function (err, data) {
      if (err) {
        console.log("Error", err);
      } else {
        console.log("Success", data);
      }
    });
  }
});
});
```

To run the example, type the following at the command line.

```
node iam_deleteuser.js USER_NAME
```

This sample 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 `AWS.IAM` client class:

- [`createPolicy`](#)
- [`getPolicy`](#)
- [`listAttachedRolePolicies`](#)
- [`attachRolePolicy`](#)
- [`detachRolePolicy`](#)

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:

- Install Node.js. For more information about installing Node.js, see the [Node.js website](#).
- 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](#).
- 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*.

Creating an IAM Policy

Create a Node.js module with the file name `iam_createpolicy.js`. Be sure to configure the SDK as previously shown. To access IAM, create an `AWS.IAM` service object. Create two JSON objects, one containing the policy document you want to create and the other containing the parameters needed to create the policy, which includes the policy JSON and the name you want to give the policy. Be sure to stringify the policy JSON object in the parameters. Call the `createPolicy` method of the `AWS.IAM` service object.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
```

```
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

var myManagedPolicy = {
  Version: "2012-10-17",
  Statement: [
    {
      Effect: "Allow",
      Action: "logs:CreateLogGroup",
      Resource: "RESOURCE_ARN",
    },
    {
      Effect: "Allow",
      Action: [
        "dynamodb:DeleteItem",
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:Scan",
        "dynamodb:UpdateItem",
      ],
      Resource: "RESOURCE_ARN",
    },
  ],
};

var params = {
  PolicyDocument: JSON.stringify(myManagedPolicy),
  PolicyName: "myDynamoDBPolicy",
};

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

To run the example, type the following at the command line.

```
node iam_createpolicy.js
```

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

Getting an IAM Policy

Create a Node.js module with the file name `iam_getpolicy.js`. Be sure to configure the SDK as previously shown. To access IAM, create an `AWS.IAM` service object. Create a JSON object containing the parameters needed retrieve a policy, which is the ARN of the policy you want to get. Call the `getPolicy` method of the `AWS.IAM` service object. Write the policy description to the console.

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

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

var params = {
  PolicyArn: "arn:aws:iam::aws:policy/AWSLambdaExecute",
};

iam.getPolicy(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.Policy.Description);
  }
});
```

To run the example, type the following at the command line.

```
node iam_getpolicy.js
```

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

Attaching a Managed Role Policy

Create a Node.js module with the file name `iam_attachrolepolicy.js`. Be sure to configure the SDK as previously shown. To access IAM, create an `AWS.IAM` service object. 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 `listAttachedRolePolicies` method of the `AWS.IAM` service object, which returns an array of managed policies to the callback function.

Check the array members to see if the policy you want to attach to the role is already attached. If the policy is not attached, call the `attachRolePolicy` method to attach it.

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

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

var paramsRoleList = {
  RoleName: process.argv[2],
};

iam.listAttachedRolePolicies(paramsRoleList, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    var myRolePolicies = data.AttachedPolicies;
    myRolePolicies.forEach(function (val, index, array) {
      if (myRolePolicies[index].PolicyName === "AmazonDynamoDBFullAccess") {
        console.log(
          "AmazonDynamoDBFullAccess is already attached to this role."
        );
        process.exit();
      }
    });
    var params = {
      PolicyArn: "arn:aws:iam::aws:policy/AmazonDynamoDBFullAccess",
      RoleName: process.argv[2],
    };
    iam.attachRolePolicy(params, function (err, data) {
      if (err) {
        console.log("Unable to attach policy to role", err);
      } else {
        console.log("Role attached successfully");
      }
    });
  }
});
```

```
});  
}  
});
```

To run the example, type the following at the command line.

```
node iam_attachrolepolicy.js IAM_ROLE_NAME
```

Detaching a Managed Role Policy

Create a Node.js module with the file name `iam_detachrolepolicy.js`. Be sure to configure the SDK as previously shown. To access IAM, create an `AWS.IAM` service object. 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 `listAttachedRolePolicies` method of the `AWS.IAM` service object, which returns an array of managed policies in the callback function.

Check the array members to see if the policy you want to detach from the role is attached. If the policy is attached, call the `detachRolePolicy` method to detach it.

```
// Load the AWS SDK for Node.js  
var AWS = require("aws-sdk");  
// Set the region  
AWS.config.update({ region: "REGION" });  
  
// Create the IAM service object  
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });  
  
var paramsRoleList = {  
  RoleName: process.argv[2],  
};  
  
iam.listAttachedRolePolicies(paramsRoleList, function (err, data) {  
  if (err) {  
    console.log("Error", err);  
  } else {  
    var myRolePolicies = data.AttachedPolicies;  
    myRolePolicies.forEach(function (val, index, array) {  
      if (myRolePolicies[index].PolicyName === "AmazonDynamoDBFullAccess") {  
        var params = {  
          PolicyArn: "arn:aws:iam::aws:policy/AmazonDynamoDBFullAccess",  

```

```
        RoleName: process.argv[2],
    };
    iam.detachRolePolicy(params, function (err, data) {
        if (err) {
            console.log("Unable to detach policy from role", err);
        } else {
            console.log("Policy detached from role successfully");
            process.exit();
        }
    });
}
});
}
```

To run the example, type the following at the command line.

```
node iam_detachrolepolicy.js IAM_ROLE_NAME
```

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 `AWS.IAM` client class:

- [createAccessKey](#)
- [listAccessKeys](#)
- [getAccessKeyLastUsed](#)
- [updateAccessKey](#)
- [deleteAccessKey](#)

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:

- Install Node.js. For more information about installing Node.js, see the [Node.js website](#).
- 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](#).

Creating Access Keys for a User

Create a Node.js module with the file name `iam_createaccesskeys.js`. Be sure to configure the SDK as previously shown. To access IAM, create an `AWS.IAM` service object. Create a JSON object containing the parameters needed to create new access keys, which includes IAM user's name. Call the `createAccessKey` method of the `AWS.IAM` service object.

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

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

iam.createAccessKey({ UserName: "IAM_USER_NAME" }, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.AccessKey);
  }
});
```

```
});
```

To run the example, type the following at the command line. 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 > newuserkeys.txt
```

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

Listing a User's Access Keys

Create a Node.js module with the file name `iam_listaccesskeys.js`. Be sure to configure the SDK as previously shown. To access IAM, create an `AWS.IAM` service object. 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 you want listed. Call the `listAccessKeys` method of the `AWS.IAM` service object.

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

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

var params = {
  MaxItems: 5,
  UserName: "IAM_USER_NAME",
};

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

To run the example, type the following at the command line.

```
node iam_listaccesskeys.js
```


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

Getting the Last Use for Access Keys

Create a Node.js module with the file name `iam_accesskeylastused.js`. Be sure to configure the SDK as previously shown. To access IAM, create an `AWS.IAM` service object. Create a JSON object containing the parameters needed to create new access keys, which is the access key ID for which you want the last use information. Call the `getAccessKeyLastUsed` method of the `AWS.IAM` service object.

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

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

iam.getAccessKeyLastUsed(
  { AccessKeyId: "ACCESS_KEY_ID" },
  function (err, data) {
    if (err) {
      console.log("Error", err);
    } else {
      console.log("Success", data.AccessKeyLastUsed);
    }
  }
);
```

To run the example, type the following at the command line.

```
node iam_accesskeylastused.js
```

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

Updating Access Key Status

Create a Node.js module with the file name `iam_updateaccesskey.js`. Be sure to configure the SDK as previously shown. To access IAM, create an `AWS.IAM` service object. 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 `AWS.IAM` service object.

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

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

var params = {
  AccessKeyId: "ACCESS_KEY_ID",
  Status: "Active",
  UserName: "USER_NAME",
};

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

To run the example, type the following at the command line.

```
node iam_updateaccesskey.js
```

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

Deleting Access Keys

Create a Node.js module with the file name `iam_deleteaccesskey.js`. Be sure to configure the SDK as previously shown. To access IAM, create an `AWS.IAM` service object. 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 `deleteAccessKey` method of the `AWS.IAM` service object.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region
```

```
AWS.config.update({ region: "REGION" });

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

var params = {
  AccessKeyId: "ACCESS_KEY_ID",
  UserName: "USER_NAME",
};

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

To run the example, type the following at the command line.

```
node iam_deleteaccesskey.js
```

This sample 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 on AWS, 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 `AWS.IAM` client class:

- [listServerCertificates](#)
- [getServerCertificate](#)
- [updateServerCertificate](#)
- [deleteServerCertificate](#)

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:

- Install Node.js. For more information about installing Node.js, see the [Node.js website](#).
- 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](#).

Listing Your Server Certificates

Create a Node.js module with the file name `iam_listservercerts.js`. Be sure to configure the SDK as previously shown. To access IAM, create an `AWS.IAM` service object. Call the `listServerCertificates` method of the `AWS.IAM` service object.

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

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

iam.listServerCertificates({}, function (err, data) {
  if (err) {
    console.log("Error", err);
  }
});
```

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

To run the example, type the following at the command line.

```
node iam_listservercerts.js
```

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

Getting a Server Certificate

Create a Node.js module with the file name `iam_getservercert.js`. Be sure to configure the SDK as previously shown. To access IAM, create an `AWS.IAM` service object. Create a JSON object containing the parameters needed get a certificate, which consists of the name of the server certificate you want. Call the `getServerCertificates` method of the `AWS.IAM` service object.

```
// Load the AWS SDK for Node.js  
var AWS = require("aws-sdk");  
// Set the region  
AWS.config.update({ region: "REGION" });  
  
// Create the IAM service object  
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });  
  
iam.getServerCertificate(  
    { ServerCertificateName: "CERTIFICATE_NAME" },  
    function (err, data) {  
        if (err) {  
            console.log("Error", err);  
        } else {  
            console.log("Success", data);  
        }  
    }  
);
```

To run the example, type the following at the command line.

```
node iam_getservercert.js
```

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

Updating a Server Certificate

Create a Node.js module with the file name `iam_updateservercert.js`. Be sure to configure the SDK as previously shown. To access IAM, create an `AWS.IAM` service object. 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 `updateServerCertificate` method of the `AWS.IAM` service object.

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

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

var params = {
  ServerCertificateName: "CERTIFICATE_NAME",
  NewServerCertificateName: "NEW_CERTIFICATE_NAME",
};

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

To run the example, type the following at the command line.

```
node iam_updateservercert.js
```

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

Deleting a Server Certificate

Create a Node.js module with the file name `iam_deleteservercert.js`. Be sure to configure the SDK as previously shown. To access IAM, create an `AWS.IAM` service object. Create a JSON object

containing the parameters needed to delete a server certificate, which consists of the name of the certificate you want to delete. Call the `deleteServerCertificates` method of the `AWS.IAM` service object.

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

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

iam.deleteServerCertificate(
  { ServerCertificateName: "CERTIFICATE_NAME" },
  function (err, data) {
    if (err) {
      console.log("Error", err);
    } else {
      console.log("Success", data);
    }
  }
);
```

To run the example, type the following at the command line.

```
node iam_deleteservercert.js
```

This sample 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 `AWS.IAM` client class:

- [createAccountAlias](#)
- [listAccountAliases](#)
- [deleteAccountAlias](#)

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:

- Install Node.js. For more information about installing Node.js, see the [Node.js website](#).
- 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](#).

Creating an Account Alias

Create a Node.js module with the file name `iam_createaccountalias.js`. Be sure to configure the SDK as previously shown. To access IAM, create an `AWS.IAM` service object. Create a JSON object containing the parameters needed to create an account alias, which includes the alias you want to create. Call the `createAccountAlias` method of the `AWS.IAM` service object.

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



```
// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

iam.createAccountAlias({ AccountAlias: process.argv[2] }, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

To run the example, type the following at the command line.

```
node iam_createaccountalias.js ALIAS
```

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

Listing Account Aliases

Create a Node.js module with the file name `iam_listaccountaliases.js`. Be sure to configure the SDK as previously shown. To access IAM, create an `AWS.IAM` service object. Create a JSON object containing the parameters needed to list account aliases, which includes the maximum number of items to return. Call the `listAccountAliases` method of the `AWS.IAM` service object.

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

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

iam.listAccountAliases({ MaxItems: 10 }, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

To run the example, type the following at the command line.

```
node iam_listaccountaliases.js
```

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

Deleting an Account Alias

Create a Node.js module with the file name `iam_deleteaccountalias.js`. Be sure to configure the SDK as previously shown. To access IAM, create an `AWS.IAM` service object. Create a JSON object containing the parameters needed to delete an account alias, which includes the alias you want deleted. Call the `deleteAccountAlias` method of the `AWS.IAM` service object.

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

// Create the IAM service object
var iam = new AWS.IAM({ apiVersion: "2010-05-08" });

iam.deleteAccountAlias({ AccountAlias: process.argv[2] }, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

To run the example, type the following at the command line.

```
node iam_deleteaccountalias.js ALIAS
```

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

Amazon Kinesis Example

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 `AWS.Kinesis` client class. For more information about using the Kinesis client class, see [Class: `AWS.Kinesis`](#) in the API reference.

Topics

- [Capturing Web Page Scroll Progress with Amazon Kinesis](#)

Capturing Web Page Scroll Progress with Amazon Kinesis



This browser script example shows:

- How to capture scroll progress in a web page with Amazon Kinesis as an example of streaming page usage metrics for later analysis.

The Scenario

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 [putRecords](#) 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.

Prerequisite Tasks

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

- Create an Kinesis stream. You need to include the stream's resource ARN in the browser script. For more information about creating Amazon Kinesis Data Streams, see [Managing Kinesis Streams](#) in the *Amazon Kinesis Data Streams Developer Guide*.
- Create 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*.
- Create an IAM role whose policy grants permission to submit data to an 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*.

Use the following role policy when creating the IAM role.

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "mobileanalytics:PutEvents",
        "cognito-sync:*"
      ],
      "Resource": [
        "*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "kinesis:Put*"
      ],
      "Resource": [
        "arn:aws:kinesis:us-east-1:111122223333:stream/stream-name"
      ]
    }
  ]
}
```

```
    }  
  ]  
}
```

The Blog Page

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 pair of `<script>` elements. One of these elements adds the SDK for JavaScript to the page and the other adds the browser script that captures scroll progress on the page and reports it to Kinesis.

```
<!DOCTYPE html>  
<html>  
  <head>  
    <title>AWS SDK for JavaScript - Amazon Kinesis Application</title>  
  </head>  
  <body>  
    <div id="BlogContent" style="width: 60%; height: 800px; overflow: auto;margin:  
auto; text-align: center;">  
      <div>  
        <p>  
          Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vestibulum  
vitae nulla eget nisl bibendum feugiat. Fusce rhoncus felis at ultricies luctus.  
Vivamus fermentum cursus sem at interdum. Proin vel lobortis nulla. Aenean rutrum  
odio in tellus semper rhoncus. Nam eu felis ac augue dapibus laoreet vel in erat.  
Vivamus vitae mollis turpis. Integer sagittis dictum odio. Duis nec sapien diam.  
In imperdiet sem nec ante laoreet, vehicula facilisis sem placerat. Duis ut metus  
egestas, ullamcorper neque et, accumsan quam. Class aptent taciti sociosqu ad litora  
torquent per conubia nostra, per inceptos himenaeos.  
        </p>  
        <!-- Additional paragraphs in the blog page appear here -->  
      </div>  
    </div>  
    <script src="https://sdk.amazonaws.com/js/aws-sdk-2.283.1.min.js"></script>  
    <script src="kinesis-example.js"></script>  
  </body>  
</html>
```

Configuring the SDK

Obtain the credentials needed to configure the SDK by calling the `CognitoIdentityCredentials` method, providing the Amazon Cognito identity pool ID. Upon success, create the Kinesis service object in the callback function.

The following code snippet shows this step. (See [Capturing Web Page Scroll Progress Code](#) for the full example.)

```
// Configure Credentials to use Cognito
AWS.config.credentials = new AWS.CognitoIdentityCredentials({
  IdentityPoolId: "IDENTITY_POOL_ID",
});

AWS.config.region = "REGION";
// We're going to partition Amazon Kinesis records based on an identity.
// We need to get credentials first, then attach our event listeners.
AWS.config.credentials.get(function (err) {
  // attach event listener
  if (err) {
    alert("Error retrieving credentials.");
    console.error(err);
    return;
  }
  // create Amazon Kinesis service object
  var kinesis = new AWS.Kinesis({
    apiVersion: "2013-12-02",
  });
```

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.

The following code snippet shows this step. (See [Capturing Web Page Scroll Progress Code](#) for the full example.)

```
// Get the ID of the Web page element.
var blogContent = document.getElementById("BlogContent");

// Get Scrollable height
```

```
var scrollableHeight = blogContent.clientHeight;

var recordData = [];
var TID = null;
blogContent.addEventListener("scroll", function (event) {
  clearTimeout(TID);
  // Prevent creating a record while a user is actively scrolling
  TID = setTimeout(function () {
    // calculate percentage
    var scrollableElement = event.target;
    var scrollHeight = scrollableElement.scrollHeight;
    var scrollTop = scrollableElement.scrollTop;

    var scrollTopPercentage = Math.round((scrollTop / scrollHeight) * 100);
    var scrollBottomPercentage = Math.round(
      ((scrollTop + scrollableHeight) / scrollHeight) * 100
    );

    // Create the Amazon Kinesis record
    var record = {
      Data: JSON.stringify({
        blog: window.location.href,
        scrollTopPercentage: scrollTopPercentage,
        scrollBottomPercentage: scrollBottomPercentage,
        time: new Date(),
      }),
      PartitionKey: "partition-" + AWS.config.credentials.identityId,
    };
    recordData.push(record);
  }, 100);
});
```

Submitting Records to Kinesis

Once each second, if there are records in the array, those pending records are sent to Kinesis.

The following code snippet shows this step. (See [Capturing Web Page Scroll Progress Code](#) for the full example.)

```
// upload data to Amazon Kinesis every second if data exists
setInterval(function () {
  if (!recordData.length) {
    return;
  }
```

```
}
// upload data to Amazon Kinesis
kinesis.putRecords(
  {
    Records: recordData,
    StreamName: "NAME_OF_STREAM",
  },
  function (err, data) {
    if (err) {
      console.error(err);
    }
  }
);
// clear record data
recordData = [];
}, 1000);
});
```

Capturing Web Page Scroll Progress Code

Here is the browser script code for the Kinesis capturing web page scroll progress example.

```
// Configure Credentials to use Cognito
AWS.config.credentials = new AWS.CognitoIdentityCredentials({
  IdentityPoolId: "IDENTITY_POOL_ID",
});

AWS.config.region = "REGION";
// We're going to partition Amazon Kinesis records based on an identity.
// We need to get credentials first, then attach our event listeners.
AWS.config.credentials.get(function (err) {
  // attach event listener
  if (err) {
    alert("Error retrieving credentials.");
    console.error(err);
    return;
  }
  // create Amazon Kinesis service object
  var kinesis = new AWS.Kinesis({
    apiVersion: "2013-12-02",
  });

  // Get the ID of the Web page element.
```



```
var blogContent = document.getElementById("BlogContent");

// Get Scrollable height
var scrollableHeight = blogContent.clientHeight;

var recordData = [];
var TID = null;
blogContent.addEventListener("scroll", function (event) {
  clearTimeout(TID);
  // Prevent creating a record while a user is actively scrolling
  TID = setTimeout(function () {
    // calculate percentage
    var scrollableElement = event.target;
    var scrollHeight = scrollableElement.scrollHeight;
    var scrollTop = scrollableElement.scrollTop;

    var scrollTopPercentage = Math.round((scrollTop / scrollHeight) * 100);
    var scrollBottomPercentage = Math.round(
      ((scrollTop + scrollableHeight) / scrollHeight) * 100
    );

    // Create the Amazon Kinesis record
    var record = {
      Data: JSON.stringify({
        blog: window.location.href,
        scrollTopPercentage: scrollTopPercentage,
        scrollBottomPercentage: scrollBottomPercentage,
        time: new Date(),
      }),
      PartitionKey: "partition-" + AWS.config.credentials.identityId,
    };
    recordData.push(record);
  }, 100);
});

// upload data to Amazon Kinesis every second if data exists
setInterval(function () {
  if (!recordData.length) {
    return;
  }
  // upload data to Amazon Kinesis
  kinesis.putRecords(
    {
      Records: recordData,
```

```
    StreamName: "NAME_OF_STREAM",
  },
  function (err, data) {
    if (err) {
      console.error(err);
    }
  }
);
// clear record data
recordData = [];
}, 1000);
});
```

Amazon S3 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.



The JavaScript API for Amazon S3 is exposed through the `AWS.S3` client class. For more information about using the Amazon S3 client class, see [Class: AWS.S3](#) in the API reference.

Topics

- [Amazon S3 Browser Examples](#)
- [Amazon S3 Node.js Examples](#)

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, etc.

Topics

- [Viewing Photos in an Amazon S3 Bucket from a Browser](#)
- [Uploading Photos to Amazon S3 from a Browser](#)

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 [listObjects](#) 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.

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 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.
- 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.
- Configure bucket permissions by following [Setting permissions for website access](#) in the *Amazon Simple Storage Service User Guide*.

Create an Identity Pool

In the [Amazon Cognito console](#), create an Amazon Cognito identity pool, as described in [the section called "Step 1: Create an Amazon Cognito Identity Pool"](#) of the *Getting Started in a Browser Script* topic.

As you create the identity pool, make note of the identity pool name, as well as the role name for the **unauthenticated** identity.

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 the **Create policy** button.
3. On the **JSON** tab, enter the following JSON definition, but replace *BUCKET_NAME* with the name of the bucket.

JSON

```
{
  "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, Create an Identity Pool. You use the policy you just created to add permissions to this identity.

Although the workflow for this task is generally the same as [the section called “Step 2: Add a Policy to the Created IAM Role”](#) of the *Getting Started in a Browser Script* topic, there are a few differences to note:

- Use the new policy that you just created, not a policy for Amazon Polly.
- On the **Attach Permissions** page, to quickly find the new policy, open the **Filter policies** list and choose **Customer managed**.

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](#) 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. In the [Amazon S3 console](#), open the bucket that you created earlier.
2. On the **Overview** tab, choose the **Create folder** button 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 the **Upload** button.
 - 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 the **Upload** button (in the lower-left corner).
4. Leave **album3** empty.

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 first `<script>` element adds the SDK to the browser script. The second `<script>` element adds the external JavaScript file that holds the browser script code.

For this example, the file is named `PhotoViewer.js`, and is located in the same folder as the HTML file. To find the current `SDK_VERSION_NUMBER`, see the API Reference for the SDK for JavaScript at [AWS SDK for JavaScript API Reference Guide](#).

```
<!DOCTYPE html>
<html>
  <head>
    <!-- **DO THIS**: -->
    <!--   Replace SDK_VERSION_NUMBER with the current SDK version number -->
    <script src="https://sdk.amazonaws.com/js/aws-sdk-SDK_VERSION_NUMBER.js"></script>
    <script src="./PhotoViewer.js"></script>
    <script>listAlbums();</script>
  </head>
  <body>
    <h1>Photo Album Viewer</h1>
    <div id="viewer" />
  </body>
</html>
```

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 `AWS.S3` service object.

```
// **DO THIS**:
//   Replace BUCKET_NAME with the bucket name.
//
var albumBucketName = "BUCKET_NAME";

// **DO THIS**:
//   Replace this block of code with the sample code located at:
//   Cognito -- Manage Identity Pools -- [identity_pool_name] -- Sample Code --
//   JavaScript
//
// Initialize the Amazon Cognito credentials provider
AWS.config.region = "REGION"; // Region
AWS.config.credentials = new AWS.CognitoIdentityCredentials({
  IdentityPoolId: "IDENTITY_POOL_ID",
});

// Create a new service object
```



```
var s3 = new AWS.S3({
  apiVersion: "2006-03-01",
  params: { Bucket: albumBucketName },
});

// A utility function to create HTML.
function getHtml(template) {
  return template.join("\n");
}
```

The rest of the 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 `listObjects` method of the `AWS.S3` 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.

```
// List the photo albums that exist in the bucket.
function listAlbums() {
  s3.listObjects({ Delimiter: "/" }, function (err, data) {
    if (err) {
      return alert("There was an error listing your albums: " + err.message);
    } else {
      var albums = data.CommonPrefixes.map(function (commonPrefix) {
        var prefix = commonPrefix.Prefix;
        var albumName = decodeURIComponent(prefix.replace("/", ""));
        return getHtml([
          "<li>",
          '<button style="margin:5px;" onclick="viewAlbum(\'\' +',
            albumName +
            '\')\>',
          albumName,
          "</button>",
          "</li>",
        ]);
      });
    }
  });
}
```

```

    });
  });
  var message = albums.length
    ? getHtml(["<p>Click on an album name to view it.</p>"])
    : "<p>You do not have any albums. Please Create album.";
  var htmlTemplate = [
    "<h2>Albums</h2>",
    message,
    "<ul>",
    getHtml(albums),
    "</ul>",
  ];
  document.getElementById("viewer").innerHTML = getHtml(htmlTemplate);
}
});
}

```

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 `AWS.S3` 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.

```

// Show the photos that exist in an album.
function viewAlbum(albumName) {
  var albumPhotosKey = encodeURIComponent(albumName) + "/";
  s3.listObjects({ Prefix: albumPhotosKey }, function (err, data) {
    if (err) {
      return alert("There was an error viewing your album: " + err.message);
    }
    // 'this' references the AWS.Request instance that represents the response
    var href = this.request.httpRequest.endpoint.href;
    var bucketUrl = href + albumBucketName + "/";

    var photos = data.Contents.map(function (photo) {
      var photoKey = photo.Key;
      var photoUrl = bucketUrl + encodeURIComponent(photoKey);
      return getHtml([
        "<span>",

```

```

        "<div>",
        "<br/>",
        '',
        "</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;");
});
}

```

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 [parent section](#) for details and prerequisites.

The HTML for the example:

```
<!DOCTYPE html>
<html>
  <head>
    <!-- **DO THIS**: -->
    <!--   Replace SDK_VERSION_NUMBER with the current SDK version number -->
    <script src="https://sdk.amazonaws.com/js/aws-sdk-SDK_VERSION_NUMBER.js"></script>
    <script src="./PhotoViewer.js"></script>
    <script>listAlbums();</script>
  </head>
  <body>
    <h1>Photo Album Viewer</h1>
    <div id="viewer" />
  </body>
</html>
```

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

The browser script code for the example:

```
//
// Data constructs and initialization.
//

// **DO THIS**:
//   Replace BUCKET_NAME with the bucket name.
//
var albumBucketName = "BUCKET_NAME";

// **DO THIS**:
//   Replace this block of code with the sample code located at:
//   Cognito -- Manage Identity Pools -- [identity_pool_name] -- Sample Code --
//   JavaScript
//
// Initialize the Amazon Cognito credentials provider
AWS.config.region = "REGION"; // Region
AWS.config.credentials = new AWS.CognitoIdentityCredentials({
```

```
    IdentityPoolId: "IDENTITY_POOL_ID",
  });

// Create a new service object
var s3 = new AWS.S3({
  apiVersion: "2006-03-01",
  params: { Bucket: albumBucketName },
});

// A utility function to create HTML.
function getHtml(template) {
  return template.join("\n");
}

//
// Functions
//

// List the photo albums that exist in the bucket.
function listAlbums() {
  s3.listObjects({ Delimiter: "/" }, function (err, data) {
    if (err) {
      return alert("There was an error listing your albums: " + err.message);
    } else {
      var albums = data.CommonPrefixes.map(function (commonPrefix) {
        var prefix = commonPrefix.Prefix;
        var albumName = decodeURIComponent(prefix.replace("/", ""));
        return getHtml([
          "<li>",
          '<button style="margin:5px;" onclick="viewAlbum(\'\' +',
            albumName +
            '\')\>',
          albumName,
          "</button>",
          "</li>",
        ]);
      });
    }
  });
  var message = albums.length
    ? getHtml(["<p>Click on an album name to view it.</p>"])
    : "<p>You do not have any albums. Please Create album.";
  var htmlTemplate = [
    "<h2>Albums</h2>",
    message,
    "<ul>",
```

```

        getHtml(albums),
        "</ul>",
    ];
    document.getElementById("viewer").innerHTML = getHtml(htmlTemplate);
}
});
}

// Show the photos that exist in an album.
function viewAlbum(albumName) {
    var albumPhotosKey = encodeURIComponent(albumName) + "/";
    s3.listObjects({ Prefix: albumPhotosKey }, function (err, data) {
        if (err) {
            return alert("There was an error viewing your album: " + err.message);
        }
        // 'this' references the AWS.Request instance that represents the response
        var href = this.request.httpRequest.endpoint.href;
        var bucketUrl = href + albumBucketName + "/";

        var photos = data.Contents.map(function (photo) {
            var photoKey = photo.Key;
            var photoUrl = bucketUrl + encodeURIComponent(photoKey);
            return getHtml([
                "<span>",
                "<div>",
                "<br/>",
                '',
                "</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;");
    });
}
```

This sample 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:

- [listObjects](#)
- [headObject](#)
- [putObject](#)
- [upload](#)
- [deleteObject](#)
- [deleteObjects](#)

Prerequisite Tasks

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

- 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 \(Federated Identities\)](#) 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.

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:DeleteObject",
        "s3:GetObject",
        "s3:ListBucket",
        "s3:PutObject",
        "s3:PutObjectAcl"
      ],
      "Resource": [
        "arn:aws:s3:::BUCKET_NAME",
        "arn:aws:s3:::BUCKET_NAME/*"
      ]
    }
  ]
}
```

Configuring CORS

Before the browser script can access the Amazon S3 bucket, you must first set up its [CORS configuration](#) as follows.

Important

In the new S3 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>
```

The Web Page

The HTML for the photo upload application consists of a `<div>` element within which the browser script creates the upload user interface. The first `<script>` element adds the SDK to the browser script. The second `<script>` element adds the external JavaScript file that holds the browser script code.

```
<!DOCTYPE html>
<html>
  <head>
    <!-- **DO THIS**: -->
    <!--   Replace SDK_VERSION_NUMBER with the current SDK version number -->
    <script src="https://sdk.amazonaws.com/js/aws-sdk-SDK_VERSION_NUMBER.js"></script>
    <script src="./s3_photoExample.js"></script>
    <script>
      function getHtml(template) {
        return template.join('\n');
      }
      listAlbums();
    </script>
  </head>
  <body>
    <h1>My Photo Albums App</h1>
    <div id="app"></div>
  </body>
</html>
```

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 `AWS.S3` service object.

```
var albumBucketName = "BUCKET_NAME";
var bucketRegion = "REGION";
var IdentityPoolId = "IDENTITY_POOL_ID";

AWS.config.update({
  region: bucketRegion,
  credentials: new AWS.CognitoIdentityCredentials({
    IdentityPoolId: IdentityPoolId,
  }),
});

var s3 = new AWS.S3({
  apiVersion: "2006-03-01",
  params: { Bucket: albumBucketName },
});
```

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 `listObjects` method of the `AWS.S3` 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.

```
function listAlbums() {
```

```

s3.listObjects({ Delimiter: "/" }, function (err, data) {
  if (err) {
    return alert("There was an error listing your albums: " + err.message);
  } else {
    var albums = data.CommonPrefixes.map(function (commonPrefix) {
      var prefix = commonPrefix.Prefix;
      var albumName = decodeURIComponent(prefix.replace("/", ""));
      return getHtml([
        "<li>",
        "<span onclick=\"deleteAlbum('\" + albumName + '\" )\">X</span>",
        "<span onclick=\"viewAlbum('\" + albumName + '\" )\">",
        albumName,
        "</span>",
        "</li>",
      ]);
    });
    var message = albums.length
      ? getHtml([
          "<p>Click on an album name to view it.</p>",
          "<p>Click on the X to delete the album.</p>",
        ])
      : "<p>You do not have any albums. Please Create album.";
    var htmlTemplate = [
      "<h2>Albums</h2>",
      message,
      "<ul>",
      getHtml(albums),
      "</ul>",
      "<button onclick=\"createAlbum(prompt('Enter Album Name:'))\">",
      "Create New Album",
      "</button>",
    ];
    document.getElementById("app").innerHTML = getHtml(htmlTemplate);
  }
});
}

```

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 `putObject` method of the `AWS.S3` service object to create the album. It then calls the `viewAlbum` function to display the new empty album.

```
function createAlbum(albumName) {
  albumName = albumName.trim();
  if (!albumName) {
    return alert("Album names must contain at least one non-space character.");
  }
  if (albumName.indexOf("/") !== -1) {
    return alert("Album names cannot contain slashes.");
  }
  var albumKey = encodeURIComponent(albumName);
  s3.headObject({ Key: albumKey }, function (err, data) {
    if (!err) {
      return alert("Album already exists.");
    }
    if (err.code !== "NotFound") {
      return alert("There was an error creating your album: " + err.message);
    }
    s3.putObject({ Key: albumKey }, function (err, data) {
      if (err) {
        return alert("There was an error creating your album: " + err.message);
      }
      alert("Successfully created album.");
      viewAlbum(albumName);
    });
  });
}
```

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 `AWS.S3` 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.

```

function viewAlbum(albumName) {
    var albumPhotosKey = encodeURIComponent(albumName) + "/";
    s3.listObjects({ Prefix: albumPhotosKey }, function (err, data) {
        if (err) {
            return alert("There was an error viewing your album: " + err.message);
        }
        // 'this' references the AWS.Response instance that represents the response
        var href = this.request.httpRequest.endpoint.href;
        var bucketUrl = href + albumBucketName + "/";

        var photos = data.Contents.map(function (photo) {
            var photoKey = photo.Key;
            var photoUrl = bucketUrl + encodeURIComponent(photoKey);
            return getHtml([
                "<span>",
                "<div>",
                '',
                "</div>",
                "<div>",
                "<span onclick=\"deletePhoto('\" +
                    albumName +
                    \"', '\" +
                    photoKey +
                    \"')\">",
                "X",
                "</span>",
                "<span>",
                photoKey.replace(albumPhotosKey, ""),
                "</span>",
                "</div>",
                "</span>",
            ]);
        });
        var message = photos.length
            ? "<p>Click on the X to delete the photo</p>"
            : "<p>You do not have any photos in this album. Please add photos.</p>";
        var htmlTemplate = [
            "<h2>",
            "Album: " + albumName,
            "</h2>",
            message,
            "<div>",
            getHtml(photos),

```

```

    "</div>",
    '<input id="photoupload" type="file" accept="image/*">',
    '<button id="addphoto" onclick="addPhoto(\'' + albumName + '\'' + "\>",
    "Add Photo",
    "</button>",
    '<button onclick="listAlbums()">',
    "Back To Albums",
    "</button>",
  ];
  document.getElementById("app").innerHTML = getHtml(htmlTemplate);
});
}

```

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 `upload` method of the Amazon S3 service object to upload the photo. After uploading the photo, the function redisplay the album so the uploaded photo appears.

```

function addPhoto(albumName) {
  var files = document.getElementById("photoupload").files;
  if (!files.length) {
    return alert("Please choose a file to upload first.");
  }
  var file = files[0];
  var fileName = file.name;
  var albumPhotosKey = encodeURIComponent(albumName) + "/";

  var photoKey = albumPhotosKey + fileName;

  // Use S3 ManagedUpload class as it supports multipart uploads
  var upload = new AWS.S3.ManagedUpload({
    params: {
      Bucket: albumBucketName,
      Key: photoKey,
      Body: file,
    },
  });

  var promise = upload.promise();
}

```



```
promise.then(
  function (data) {
    alert("Successfully uploaded photo.");
    viewAlbum(albumName);
  },
  function (err) {
    return alert("There was an error uploading your photo: ", err.message);
  }
);
}
```

Deleting a Photo

To delete a photo from an album in the Amazon S3 bucket, the application's `deletePhoto` function calls the `deleteObject` method of the Amazon S3 service object. This deletes the photo specified by the `photoKey` value passed to the function.

```
function deletePhoto(albumName, photoKey) {
  s3.deleteObject({ Key: photoKey }, function (err, data) {
    if (err) {
      return alert("There was an error deleting your photo: ", err.message);
    }
    alert("Successfully deleted photo.");
    viewAlbum(albumName);
  });
}
```

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 service object.

```
function deleteAlbum(albumName) {
  var albumKey = encodeURIComponent(albumName) + "/";
  s3.listObjects({ Prefix: albumKey }, function (err, data) {
    if (err) {
      return alert("There was an error deleting your album: ", err.message);
    }
    var objects = data.Contents.map(function (object) {
      return { Key: object.Key };
    });
  });
}
```

```
s3.deleteObjects(  
  {  
    Delete: { Objects: objects, Quiet: true },  
  },  
  function (err, data) {  
    if (err) {  
      return alert("There was an error deleting your album: ", err.message);  
    }  
    alert("Successfully deleted album.");  
    listAlbums();  
  }  
);  
});  
}
```

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](#) for details and prerequisites.

The HTML for the example:

```
<!DOCTYPE html>  
<html>  
  <head>  
    <!-- **DO THIS**: -->  
    <!-- Replace SDK_VERSION_NUMBER with the current SDK version number -->  
    <script src="https://sdk.amazonaws.com/js/aws-sdk-SDK_VERSION_NUMBER.js"></script>  
    <script src="./s3_photoExample.js"></script>  
    <script>  
      function getHtml(template) {  
        return template.join('\n');  
      }  
      listAlbums();  
    </script>  
  </head>  
  <body>  
    <h1>My Photo Albums App</h1>  
    <div id="app"></div>  
  </body>  
</html>
```

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

The browser script code for the example:

```
var albumBucketName = "BUCKET_NAME";
var bucketRegion = "REGION";
var IdentityPoolId = "IDENTITY_POOL_ID";

AWS.config.update({
  region: bucketRegion,
  credentials: new AWS.CognitoIdentityCredentials({
    IdentityPoolId: IdentityPoolId,
  }),
});

var s3 = new AWS.S3({
  apiVersion: "2006-03-01",
  params: { Bucket: albumBucketName },
});

function listAlbums() {
  s3.listObjects({ Delimiter: "/" }, function (err, data) {
    if (err) {
      return alert("There was an error listing your albums: " + err.message);
    } else {
      var albums = data.CommonPrefixes.map(function (commonPrefix) {
        var prefix = commonPrefix.Prefix;
        var albumName = decodeURIComponent(prefix.replace("/", ""));
        return getHtml([
          "<li>",
          "<span onclick=\"deleteAlbum('" + albumName + "')\">X</span>",
          "<span onclick=\"viewAlbum('" + albumName + "')\">",
          albumName,
          "</span>",
          "</li>",
        ]);
      });
      var message = albums.length
        ? getHtml([
            "<p>Click on an album name to view it.</p>",
            "<p>Click on the X to delete the album.</p>",
          ])
        : "<p>You do not have any albums. Please Create album.";
      var htmlTemplate = [
        "<h2>Albums</h2>",
        message,
      ];
    }
  });
}
```

```

        "<ul>",
        getHtml(albums),
        "</ul>",
        "<button onclick=\"createAlbum(prompt('Enter Album Name:'))\">",
        "Create New Album",
        "</button>",
    ];
    document.getElementById("app").innerHTML = getHtml(htmlTemplate);
}
});
}

function createAlbum(albumName) {
    albumName = albumName.trim();
    if (!albumName) {
        return alert("Album names must contain at least one non-space character.");
    }
    if (albumName.indexOf("/") !== -1) {
        return alert("Album names cannot contain slashes.");
    }
    var albumKey = encodeURIComponent(albumName);
    s3.headObject({ Key: albumKey }, function (err, data) {
        if (!err) {
            return alert("Album already exists.");
        }
        if (err.code !== "NotFound") {
            return alert("There was an error creating your album: " + err.message);
        }
        s3.putObject({ Key: albumKey }, function (err, data) {
            if (err) {
                return alert("There was an error creating your album: " + err.message);
            }
            alert("Successfully created album.");
            viewAlbum(albumName);
        });
    });
}

function viewAlbum(albumName) {
    var albumPhotosKey = encodeURIComponent(albumName) + "/";
    s3.listObjects({ Prefix: albumPhotosKey }, function (err, data) {
        if (err) {
            return alert("There was an error viewing your album: " + err.message);
        }
    });
}

```

```

// 'this' references the AWS.Response instance that represents the response
var href = this.request.httpRequest.endpoint.href;
var bucketUrl = href + albumBucketName + "/";

var photos = data.Contents.map(function (photo) {
  var photoKey = photo.Key;
  var photoUrl = bucketUrl + encodeURIComponent(photoKey);
  return getHtml([
    "<span>",
    "<div>",
    '',
    "</div>",
    "<div>",
    "<span onclick=\"deletePhoto('\" +
      albumName +
      \"', '\" +
      photoKey +
      \"')\">",
    "X",
    "</span>",
    "<span>",
    photoKey.replace(albumPhotosKey, ""),
    "</span>",
    "</div>",
    "</span>",
  ]);
});
var message = photos.length
  ? "<p>Click on the X to delete the photo</p>"
  : "<p>You do not have any photos in this album. Please add photos.</p>";
var htmlTemplate = [
  "<h2>",
  "Album: " + albumName,
  "</h2>",
  message,
  "<div>",
  getHtml(photos),
  "</div>",
  '<input id="photoupload" type="file" accept="image/*">',
  '<button id="addphoto" onclick="addPhoto(\'\" + albumName + \"')\">',
  "Add Photo",
  "</button>",
  '<button onclick="listAlbums()">',
  "Back To Albums",

```

```
        "</button>",
    ];
    document.getElementById("app").innerHTML = getHtml(htmlTemplate);
});
}

function addPhoto(albumName) {
    var files = document.getElementById("photoupload").files;
    if (!files.length) {
        return alert("Please choose a file to upload first.");
    }
    var file = files[0];
    var fileName = file.name;
    var albumPhotosKey = encodeURIComponent(albumName) + "/";

    var photoKey = albumPhotosKey + fileName;

    // Use S3 ManagedUpload class as it supports multipart uploads
    var upload = new AWS.S3.ManagedUpload({
        params: {
            Bucket: albumBucketName,
            Key: photoKey,
            Body: file,
        },
    });

    var promise = upload.promise();

    promise.then(
        function (data) {
            alert("Successfully uploaded photo.");
            viewAlbum(albumName);
        },
        function (err) {
            return alert("There was an error uploading your photo: ", err.message);
        }
    );
}

function deletePhoto(albumName, photoKey) {
    s3.deleteObject({ Key: photoKey }, function (err, data) {
        if (err) {
            return alert("There was an error deleting your photo: ", err.message);
        }
    });
}
```

```
    alert("Successfully deleted photo.");
    viewAlbum(albumName);
  });
}

function deleteAlbum(albumName) {
  var albumKey = encodeURIComponent(albumName) + "/";
  s3.listObjects({ Prefix: albumKey }, function (err, data) {
    if (err) {
      return alert("There was an error deleting your album: ", err.message);
    }
    var objects = data.Contents.map(function (object) {
      return { Key: object.Key };
    });
    s3.deleteObjects(
      {
        Delete: { Objects: objects, Quiet: true },
      },
      function (err, data) {
        if (err) {
          return alert("There was an error deleting your album: ", err.message);
        }
        alert("Successfully deleted album.");
        listAlbums();
      }
    );
  });
}
```

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](#)
- [Configuring Amazon S3 Buckets](#)
- [Managing Amazon S3 Bucket Access Permissions](#)
- [Working with Amazon S3 Bucket Policies](#)

- [Using an Amazon S3 Bucket as a Static Web Host](#)

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:

- [listBuckets](#)
- [createBucket](#)
- [listObjects](#)
- [upload](#)
- [deleteBucket](#)

Prerequisite Tasks

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

- Install Node.js. For more information about installing Node.js, see the [Node.js website](#).
- 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](#).

Configuring the SDK

Configure the SDK for JavaScript by creating a global configuration object then setting the Region for your code. In this example, the Region is set to `us-west-2`.

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

Displaying a List of Amazon S3 Buckets

Create a Node.js module with the file name `s3_listbuckets.js`. Make sure to configure the SDK as previously shown. To access Amazon Simple Storage Service, create an `AWS.S3` service object. Call the `listBuckets` method of the Amazon S3 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.

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

// Create S3 service object
s3 = new AWS.S3({ apiVersion: "2006-03-01" });

// Call S3 to list the buckets
s3.listBuckets(function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.Buckets);
  }
});
```

To run the example, type the following at the command line.

```
node s3_listbuckets.js
```

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

Creating an Amazon S3 Bucket

Create a Node.js module with the file name `s3_createbucket.js`. Make sure to configure the SDK as previously shown. Create an AWS .S3 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 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.

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

// Create S3 service object
s3 = new AWS.S3({ apiVersion: "2006-03-01" });

// Create the parameters for calling createBucket
var bucketParams = {
  Bucket: process.argv[2],
};

// call S3 to create the bucket
s3.createBucket(bucketParams, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.Location);
  }
});
```

To run the example, type the following at the command line.

```
node s3_createbucket.js BUCKET_NAME
```

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

Uploading a File to an Amazon S3 Bucket

Create a Node.js module with the file name `s3_upload.js`. Make sure to configure the SDK as previously shown. Create an AWS .S3 service object. The module will take two command-line

arguments, the first one to specify the destination bucket and the second to specify the file to upload.

Create a variable with the parameters needed to call the `upload` 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 selected file, which you can obtain using the Node.js `path` module. The `Body` parameter is set to the contents of the file, which you can obtain using `createReadStream` from the Node.js `fs` module.

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

// Create S3 service object
var s3 = new AWS.S3({ apiVersion: "2006-03-01" });

// call S3 to retrieve upload file to specified bucket
var uploadParams = { Bucket: process.argv[2], Key: "", Body: "" };
var file = process.argv[3];

// Configure the file stream and obtain the upload parameters
var fs = require("fs");
var fileStream = fs.createReadStream(file);
fileStream.on("error", function (err) {
    console.log("File Error", err);
});
uploadParams.Body = fileStream;
var path = require("path");
uploadParams.Key = path.basename(file);

// call S3 to retrieve upload file to specified bucket
s3.upload(uploadParams, function (err, data) {
    if (err) {
        console.log("Error", err);
    }
    if (data) {
        console.log("Upload Success", data.Location);
    }
});
```

To run the example, type the following at the command line.

```
node s3_upload.js BUCKET_NAME FILE_NAME
```

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

Listing Objects in an Amazon S3 Bucket

Create a Node.js module with the file name `s3_listobjects.js`. Make sure to configure the SDK as previously shown. Create an `AWS.S3` service object.

Add a variable to hold the parameters used to call the `listObjects` 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.

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

// Create S3 service object
s3 = new AWS.S3({ apiVersion: "2006-03-01" });

// Create the parameters for calling listObjects
var bucketParams = {
  Bucket: "BUCKET_NAME",
};

// Call S3 to obtain a list of the objects in the bucket
s3.listObjects(bucketParams, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

To run the example, type the following at the command line.

```
node s3_listobjects.js
```

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

Deleting an Amazon S3 Bucket

Create a Node.js module with the file name `s3_deletebucket.js`. Make sure to configure the SDK as previously shown. Create an `AWS.S3` service object.

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 in order to delete it. The callback function logs a success or failure message.

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

// Create S3 service object
s3 = new AWS.S3({ apiVersion: "2006-03-01" });

// Create params for S3.deleteBucket
var bucketParams = {
    Bucket: "BUCKET_NAME",
};

// Call S3 to delete the bucket
s3.deleteBucket(bucketParams, function (err, data) {
    if (err) {
        console.log("Error", err);
    } else {
        console.log("Success", data);
    }
});
```

To run the example, type the following at the command line.

```
node s3_deletebucket.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:

- [getBucketCors](#)
- [putBucketCors](#)

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:

- Install Node.js. For more information about installing Node.js, see the [Node.js website](#).
- 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](#).

Configuring the SDK

Configure the SDK for JavaScript by creating a global configuration object then setting the Region for your code. In this example, the Region is set to `us-west-2`.

```
// Load the SDK for JavaScript
```

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

Retrieving a Bucket CORS Configuration

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. Create an `AWS.S3` service object.

The only parameter you need to pass is the name of the selected bucket when calling the `getBucketCors` 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.

```
// Load the AWS SDK for Node.js  
var AWS = require("aws-sdk");  
// Set the region  
AWS.config.update({ region: "REGION" });  
  
// Create S3 service object  
s3 = new AWS.S3({ apiVersion: "2006-03-01" });  
  
// Set the parameters for S3.getBucketCors  
var bucketParams = { Bucket: process.argv[2] };  
  
// call S3 to retrieve CORS configuration for selected bucket  
s3.getBucketCors(bucketParams, function (err, data) {  
  if (err) {  
    console.log("Error", err);  
  } else if (data) {  
    console.log("Success", JSON.stringify(data.CORSRules));  
  }  
});
```

To run the example, type the following at the command line.

```
node s3_getcors.js BUCKET_NAME
```

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

Setting a Bucket CORS Configuration

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.

Create an `AWS.S3` service object. Next create a JSON object to hold the values for the CORS configuration as required by the `putBucketCors` method of the `AWS.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.

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

// Create S3 service object
s3 = new AWS.S3({ apiVersion: "2006-03-01" });

// Create initial parameters JSON for putBucketCors
var thisConfig = {
  AllowedHeaders: ["Authorization"],
  AllowedMethods: [],
  AllowedOrigins: ["*"],
  ExposeHeaders: [],
  MaxAgeSeconds: 3000,
};

// Assemble the list of allowed methods based on command line parameters
var allowedMethods = [];
process.argv.forEach(function (val, index, array) {
  if (val.toUpperCase() === "POST") {
    allowedMethods.push("POST");
  }
});
```



```
    if (val.toUpperCase() === "GET") {
      allowedMethods.push("GET");
    }
    if (val.toUpperCase() === "PUT") {
      allowedMethods.push("PUT");
    }
    if (val.toUpperCase() === "PATCH") {
      allowedMethods.push("PATCH");
    }
    if (val.toUpperCase() === "DELETE") {
      allowedMethods.push("DELETE");
    }
    if (val.toUpperCase() === "HEAD") {
      allowedMethods.push("HEAD");
    }
  });

  // Copy the array of allowed methods into the config object
  thisConfig.AllowedMethods = allowedMethods;
  // Create array of configs then add the config object to it
  var corsRules = new Array(thisConfig);

  // Create CORS params
  var corsParams = {
    Bucket: process.argv[2],
    CORSConfiguration: { CORSRules: corsRules },
  };

  // set the new CORS configuration on the selected bucket
  s3.putBucketCors(corsParams, function (err, data) {
    if (err) {
      // display error message
      console.log("Error", err);
    } else {
      // update the displayed CORS config for the selected bucket
      console.log("Success", data);
    }
  });
});
```

To run the example, type the following at the command line including one or more HTTP methods as shown.

```
node s3_setcors.js BUCKET_NAME get put
```

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:

- [getBucketAcl](#)
- [putBucketAcl](#)

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:

- Install Node.js. For more information about installing Node.js, see the [Node.js website](#).
- 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](#).

Configuring the SDK

Configure the SDK for JavaScript by creating a global configuration object then setting the Region for your code. In this example, the Region is set to us-west-2.

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

Retrieving the Current Bucket Access Control List

Create a Node.js module with the file name `s3_getbucketacl.js`. The module will take a single command-line argument to specify the bucket whose ACL configuration you want. Make sure to configure the SDK as previously shown.

Create an `AWS.S3` service object. The only parameter you need to pass is the name of the selected bucket when calling the `getBucketAcl` method. The current access control list configuration is returned by Amazon S3 in the `data` parameter passed to the callback function.

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

// Create S3 service object
s3 = new AWS.S3({ apiVersion: "2006-03-01" });

var bucketParams = { Bucket: process.argv[2] };
// call S3 to retrieve policy for selected bucket
s3.getBucketAcl(bucketParams, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else if (data) {
    console.log("Success", data.Grants);
  }
});
```

To run the example, type the following at the command line.

```
node s3_getbucketacl.js BUCKET_NAME
```

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:

- [getBucketPolicy](#)
- [putBucketPolicy](#)
- [deleteBucketPolicy](#)

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:

- Install Node.js. For more information about installing Node.js, see the [Node.js website](#).
- 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](#).

Configuring the SDK

Configure the SDK for JavaScript by creating a global configuration object then setting the Region for your code. In this example, the Region is set to `us-west-2`.

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

Retrieving the Current Bucket Policy

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.

Create an `AWS.S3` service object. The only parameter you need to pass is the name of the selected bucket when calling the `getBucketPolicy` 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.

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

// Create S3 service object
s3 = new AWS.S3({ apiVersion: "2006-03-01" });

var bucketParams = { Bucket: process.argv[2] };
// call S3 to retrieve policy for selected bucket
s3.getBucketPolicy(bucketParams, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else if (data) {
    console.log("Success", data.Policy);
  }
});
```

To run the example, type the following at the command line.

```
node s3_getbucketpolicy.js BUCKET_NAME
```

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

Setting a Simple Bucket Policy

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.

Create an `AWS.S3` service object. 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 `putBucketPolicy` method, including the name of the bucket and the JSON policy converted to a string value.

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

// Create S3 service object
s3 = new AWS.S3({ apiVersion: "2006-03-01" });

var readOnlyAnonUserPolicy = {
  Version: "2012-10-17",
  Statement: [
    {
      Sid: "AddPerm",
      Effect: "Allow",
      Principal: "*",
      Action: ["s3:GetObject"],
      Resource: [""],
    },
  ],
};

// create selected bucket resource string for bucket policy
var bucketResource = "arn:aws:s3:::" + process.argv[2] + "/*";
readOnlyAnonUserPolicy.Statement[0].Resource[0] = bucketResource;
```

```
// convert policy JSON into string and assign into params
var bucketPolicyParams = {
  Bucket: process.argv[2],
  Policy: JSON.stringify(readOnlyAnonUserPolicy),
};

// set the new policy on the selected bucket
s3.putBucketPolicy(bucketPolicyParams, function (err, data) {
  if (err) {
    // display error message
    console.log("Error", err);
  } else {
    console.log("Success", data);
  }
});
```

To run the example, type the following at the command line.

```
node s3_setbucketpolicy.js BUCKET_NAME
```

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

Deleting a Bucket Policy

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.

Create an `AWS.S3` service object. The only parameter you need to pass when calling the `deleteBucketPolicy` method is the name of the selected bucket.

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

// Create S3 service object
s3 = new AWS.S3({ apiVersion: "2006-03-01" });

var bucketParams = { Bucket: process.argv[2] };
// call S3 to delete policy for selected bucket
s3.deleteBucketPolicy(bucketParams, function (err, data) {
```

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

To run the example, type the following at the command line.

```
node s3_deletebucketpolicy.js BUCKET_NAME
```

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:

- [getBucketWebsite](#)
- [putBucketWebsite](#)
- [deleteBucketWebsite](#)

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:

- Install Node.js. For more information about installing Node.js, see the [Node.js website](#).
- 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](#).

Configuring the SDK

Configure the SDK for JavaScript by creating a global configuration object then setting the Region for your code. In this example, the Region is set to `us-west-2`.

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

Retrieving the Current Bucket Website Configuration

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.

Create an `AWS.S3` service object. 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 `getBucketWebsite` 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.

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

// Create S3 service object
s3 = new AWS.S3({ apiVersion: "2006-03-01" });

var bucketParams = { Bucket: process.argv[2] };
```

```
// call S3 to retrieve the website configuration for selected bucket
s3.getBucketWebsite(bucketParams, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else if (data) {
    console.log("Success", data);
  }
});
```

To run the example, type the following at the command line.

```
node s3_getbucketwebsite.js BUCKET_NAME
```

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

Setting a Bucket Website Configuration

Create a Node.js module with the file name `s3_setbucketwebsite.js`. Make sure to configure the SDK as previously shown. Create an AWS.S3 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 `putBucketWebsite` method, including the name of the bucket and the JSON website configuration.

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

// Create S3 service object
s3 = new AWS.S3({ apiVersion: "2006-03-01" });

// Create JSON for putBucketWebsite parameters
var staticHostParams = {
  Bucket: "",
  WebsiteConfiguration: {
```

```
    ErrorDocument: {
      Key: "",
    },
    IndexDocument: {
      Suffix: "",
    },
  },
};

// Insert specified bucket name and index and error documents into params JSON
// from command line arguments
staticHostParams.Bucket = process.argv[2];
staticHostParams.WebsiteConfiguration.IndexDocument.Suffix = process.argv[3];
staticHostParams.WebsiteConfiguration.ErrorDocument.Key = process.argv[4];

// set the new website configuration on the selected bucket
s3.putBucketWebsite(staticHostParams, function (err, data) {
  if (err) {
    // display error message
    console.log("Error", err);
  } else {
    // update the displayed website configuration for the selected bucket
    console.log("Success", data);
  }
});
```

To run the example, type the following at the command line.

```
node s3_setbucketwebsite.js BUCKET_NAME INDEX_PAGE ERROR_PAGE
```

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

Deleting a Bucket Website Configuration

Create a Node.js module with the file name `s3_deletebucketwebsite.js`. Make sure to configure the SDK as previously shown. Create an `AWS.S3` service object.

Create a function that deletes the website configuration for the selected bucket. The only parameter you need to pass when calling the `deleteBucketWebsite` method is the name of the selected bucket.

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

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

// Create S3 service object
s3 = new AWS.S3({ apiVersion: "2006-03-01" });

var bucketParams = { Bucket: process.argv[2] };

// call S3 to delete website configuration for selected bucket
s3.deleteBucketWebsite(bucketParams, function (error, data) {
  if (error) {
    console.log("Error", err);
  } else if (data) {
    console.log("Success", data);
  }
});
```

To run the example, type the following at the command line.

```
node s3_deletebucketwebsite.js BUCKET_NAME
```

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 `AWS.SES` client class. For more information about using the Amazon SES client class, see [Class: `AWS.SES`](#) in the API reference.

Topics

- [Managing Amazon SES Identities](#)
- [Working with Email Templates in Amazon SES](#)
- [Sending Email Using Amazon SES](#)
- [Using IP Address Filters for Email Receipt in Amazon SES](#)
- [Using Receipt Rules in Amazon SES](#)

Managing Amazon SES Identities



This Node.js code example shows:

- How to verify email addresses and domains used with Amazon SES.
- How to assign 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 `AWS.SES` client class:

- [listIdentities](#)
- [deleteIdentity](#)
- [verifyEmailIdentity](#)
- [verifyDomainIdentity](#)

Prerequisite Tasks

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

- Install Node.js. For more information about installing Node.js, see the [Node.js website](#).
- 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](#).

Configuring the SDK

Configure the SDK for JavaScript by creating a global configuration object then setting the Region for your code. In this example, the Region is set to `us-west-2`.

```
// Load the SDK for JavaScript
var AWS = require('aws-sdk');

// Set the Region
AWS.config.update({region: 'us-west-2'});
```

Listing Your Identities

In this example, use a Node.js module to list email addresses and domains to use with Amazon SES. Create a Node.js module with the file name `ses_listidentities.js`. Configure the SDK as previously shown.

Create an object to pass the `IdentityType` and other parameters for the `listIdentities` method of the `AWS.SES` client class. To call the `listIdentities` method, create a promise for invoking an Amazon SES service object, passing the parameters object.

Then handle the response in the promise callback. The data returned by the promise contains an array of domain identities as specified by the `IdentityType` parameter.

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

// Create listIdentities params
var params = {
  IdentityType: "Domain",
  MaxItems: 10,
};

// Create the promise and SES service object
var listIDsPromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .listIdentities(params)
  .promise();

// Handle promise's fulfilled/rejected states
listIDsPromise
  .then(function (data) {
    console.log(data.Identities);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

To run the example, type the following at the command line.

```
node ses_listidentities.js
```

This sample 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 Node.js module with the file name `ses_verifyemailidentity.js`. Configure the SDK as previously shown. To access Amazon SES, create an `AWS.SES` service object.

Create an object to pass the `EmailAddress` parameter for the `verifyEmailIdentity` method of the `AWS.SES` client class. To call the `verifyEmailIdentity` method, create a promise for invoking an Amazon SES service object, passing the parameters. Then handle the response in the promise callback.

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

// Create promise and SES service object
var verifyEmailPromise = new AWS.SES({ apiVersion: "2010-12-01" })
    .verifyEmailIdentity({ EmailAddress: "ADDRESS@DOMAIN.EXT" })
    .promise();

// Handle promise's fulfilled/rejected states
verifyEmailPromise
    .then(function (data) {
        console.log("Email verification initiated");
    })
    .catch(function (err) {
        console.error(err, err.stack);
    });
```

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

```
node ses_verifyemailidentity.js
```

This sample 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 Node.js module with the file name `ses_verifydomainidentity.js`. Configure the SDK as previously shown.

Create an object to pass the `Domain` parameter for the `verifyDomainIdentity` method of the `AWS.SES` client class. To call the `verifyDomainIdentity` method, create a promise for invoking an Amazon SES service object, passing the parameters object. Then handle the response in the promise callback.

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

// Create the promise and SES service object
var verifyDomainPromise = new AWS.SES({ apiVersion: "2010-12-01" })
    .verifyDomainIdentity({ Domain: "DOMAIN_NAME" })
    .promise();

// Handle promise's fulfilled/rejected states
verifyDomainPromise
    .then(function (data) {
        console.log("Verification Token: " + data.VerificationToken);
    })
    .catch(function (err) {
        console.error(err, err.stack);
    });
```

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

```
node ses_verifydomainidentity.js
```

This sample 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 Node.js module with the file name `ses_deleteidentity.js`. Configure the SDK as previously shown.

Create an object to pass the `Identity` parameter for the `deleteIdentity` method of the `AWS.SES` client class. To call the `deleteIdentity` method, create a request for invoking an Amazon SES service object, passing the parameters. Then handle the response in the promise callback..

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

// Create the promise and SES service object
var deletePromise = new AWS.SES({ apiVersion: "2010-12-01" })
    .deleteIdentity({ Identity: "DOMAIN_NAME" })
    .promise();

// Handle promise's fulfilled/rejected states
deletePromise
    .then(function (data) {
        console.log("Identity Deleted");
    })
    .catch(function (err) {
        console.error(err, err.stack);
    });
```

To run the example, type the following at the command line.

```
node ses_deleteidentity.js
```

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

Working with Email Templates in Amazon SES



This Node.js code example shows:

- Get a list of all of your email templates.
- Retrieve and update email templates.
- Create and delete email templates.

Amazon SES lets you send personalized email messages using email templates. For details on how to create and use email templates in Amazon Simple Email Service, 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 `AWS.SES` client class:

- [listTemplates](#)
- [createTemplate](#)
- [getTemplate](#)
- [deleteTemplate](#)
- [updateTemplate](#)

Prerequisite Tasks

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

- Install Node.js. For more information about installing Node.js, see the [Node.js website](#).
- 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](#).

Listing Your Email Templates

In this example, use a Node.js module to create an email template to use with Amazon SES. Create a Node.js module with the file name `ses_listtemplates.js`. Configure the SDK as previously shown.

Create an object to pass the parameters for the `listTemplates` method of the `AWS.SES` client class. To call the `listTemplates` method, create a promise for invoking an Amazon SES service object, passing the parameters. Then handle the response in the promise callback.

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

// Create the promise and SES service object
var templatePromise = new AWS.SES({ apiVersion: "2010-12-01" })
    .listTemplates({ MaxItems: ITEMS_COUNT })
    .promise();

// Handle promise's fulfilled/rejected states
templatePromise
    .then(function (data) {
        console.log(data);
    })
    .catch(function (err) {
        console.error(err, err.stack);
    });
```

To run the example, type the following at the command line. Amazon SES returns the list of templates.

```
node ses_listtemplates.js
```

This sample 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 Node.js module with the file name `ses_gettemplate.js`. Configure the SDK as previously shown.

Create an object to pass the `TemplateName` parameter for the `getTemplate` method of the `AWS.SES` client class. To call the `getTemplate` method, create a promise for invoking an Amazon SES service object, passing the parameters. Then handle the response in the promise callback.

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

// Create the promise and Amazon Simple Email Service (Amazon SES) service object.
var templatePromise = new AWS.SES({ apiVersion: "2010-12-01" })
    .getTemplate({ TemplateName: "TEMPLATE_NAME" })
    .promise();

// Handle promise's fulfilled/rejected states
templatePromise
    .then(function (data) {
        console.log(data.Template.SubjectPart);
    })
    .catch(function (err) {
        console.error(err, err.stack);
    });
```

To run the example, type the following at the command line. Amazon SES returns the template details.

```
node ses_gettemplate.js
```

This sample 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 Node.js module with the file name `ses_createtemplate.js`. Configure the SDK as previously shown.

Create an object to pass the parameters for the `createTemplate` method of the `AWS.SES` client class, including `TemplateName`, `HtmlPart`, `SubjectPart`, and `TextPart`. To call the `createTemplate` method, create a promise for invoking an Amazon SES service object, passing the parameters. Then handle the response in the promise callback.

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

// Create createTemplate params
var params = {
  Template: {
    TemplateName: "TEMPLATE_NAME" /* required */,
    HtmlPart: "HTML_CONTENT",
    SubjectPart: "SUBJECT_LINE",
    TextPart: "TEXT_CONTENT",
  },
};

// Create the promise and SES service object
var templatePromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .createTemplate(params)
  .promise();

// Handle promise's fulfilled/rejected states
templatePromise
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

To run the example, type the following at the command line. The template is added to Amazon SES.

```
node ses_createtemplate.js
```

This sample 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 Node.js module with the file name `ses_updatetemplate.js`. Configure the SDK as previously shown.

Create an object to pass the Template parameter values you want to update in the template, with the required `TemplateName` parameter passed to the `updateTemplate` method of the `AWS.SES` client class. To call the `updateTemplate` method, create a promise for invoking an Amazon SES service object, passing the parameters. Then handle the response in the promise callback.

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

// Create updateTemplate parameters
var params = {
  Template: {
    TemplateName: "TEMPLATE_NAME" /* required */,
    HtmlPart: "HTML_CONTENT",
    SubjectPart: "SUBJECT_LINE",
    TextPart: "TEXT_CONTENT",
  },
};

// Create the promise and SES service object
var templatePromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .updateTemplate(params)
  .promise();

// Handle promise's fulfilled/rejected states
templatePromise
  .then(function (data) {
    console.log("Template Updated");
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

To run the example, type the following at the command line. Amazon SES returns the template details.

```
node ses_updatetemplate.js
```

This sample 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 Node.js module with the file name `ses_deletetemplate.js`. Configure the SDK as previously shown.

Create an object to pass the `requiredTemplateName` parameter to the `deleteTemplate` method of the `AWS.SES` client class. To call the `deleteTemplate` method, create a promise for invoking an Amazon SES service object, passing the parameters. Then handle the response in the promise callback.

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

// Create the promise and SES service object
var templatePromise = new AWS.SES({ apiVersion: "2010-12-01" })
    .deleteTemplate({ TemplateName: "TEMPLATE_NAME" })
    .promise();

// Handle promise's fulfilled/rejected states
templatePromise
    .then(function (data) {
        console.log("Template Deleted");
    })
    .catch(function (err) {
        console.error(err, err.stack);
    });
```

To run the example, type the following at the command line. Amazon SES returns the template details.

```
node ses_deletetemplate.js
```

This sample 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 `AWS.SES` client class:

- [sendEmail](#)
- [sendTemplatedEmail](#)
- [sendBulkTemplatedEmail](#)

Prerequisite Tasks

- Install Node.js. For more information about installing Node.js, see the [Node.js website](#).
- 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](#).

Email Message Sending Requirements

Amazon SES composes an email message and immediately queues it for sending. To send email using the `SES.sendEmail` 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 invalid (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 Node.js module with the file name `ses_sendemail.js`. Configure the SDK as previously shown.

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 `sendEmail` method of the `AWS.SES` client class. To call the `sendEmail` method, create a promise for invoking an Amazon SES service object, passing the parameters. Then handle the response in the promise callback.

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

```
// Create sendEmail params
var params = {
  Destination: {
    /* required */
    CcAddresses: [
      "EMAIL_ADDRESS",
      /* more items */
    ],
    ToAddresses: [
      "EMAIL_ADDRESS",
      /* more items */
    ],
  },
  Message: {
    /* required */
    Body: {
      /* required */
      Html: {
        Charset: "UTF-8",
        Data: "HTML_FORMAT_BODY",
      },
      Text: {
        Charset: "UTF-8",
        Data: "TEXT_FORMAT_BODY",
      },
    },
    Subject: {
      Charset: "UTF-8",
      Data: "Test email",
    },
  },
  Source: "SENDER_EMAIL_ADDRESS" /* required */,
  ReplyToAddresses: [
    "EMAIL_ADDRESS",
    /* more items */
  ],
};

// Create the promise and SES service object
var sendPromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .sendEmail(params)
  .promise();

// Handle promise's fulfilled/rejected states
```

```
sendPromise
  .then(function (data) {
    console.log(data.MessageId);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

To run the example, type the following at the command line. The email is queued for sending by Amazon SES.

```
node ses_sendemail.js
```

This sample code can be [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.

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 `sendTemplatedEmail` method of the `AWS.SES` client class. To call the `sendTemplatedEmail` method, create a promise for invoking an Amazon SES service object, passing the parameters. Then handle the response in the promise callback.

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

// Create sendTemplatedEmail params
var params = {
  Destination: {
    /* required */
    CcAddresses: [
      "EMAIL_ADDRESS",
      /* more CC email addresses */
    ],
    ToAddresses: [
      "EMAIL_ADDRESS",
      /* more To email addresses */
    ],
  },
};
```

```
    ],  
  },  
  Source: "EMAIL_ADDRESS" /* required */,  
  Template: "TEMPLATE_NAME" /* required */,  
  TemplateData: '{ "REPLACEMENT_TAG_NAME":"REPLACEMENT_VALUE" }' /* required */,  
  ReplyToAddresses: ["EMAIL_ADDRESS"],  
};  
  
// Create the promise and SES service object  
var sendPromise = new AWS.SES({ apiVersion: "2010-12-01" })  
  .sendTemplatedEmail(params)  
  .promise();  
  
// Handle promise's fulfilled/rejected states  
sendPromise  
  .then(function (data) {  
    console.log(data);  
  })  
  .catch(function (err) {  
    console.error(err, err.stack);  
  });
```

To run the example, type the following at the command line. The email is queued for sending by Amazon SES.

```
node ses_sendtemplatedemail.js
```

This sample 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 Node.js module with the file name `ses_sendbulktemplatedemail.js`. Configure the SDK as previously shown.

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 `sendBulkTemplatedEmail` method of the `AWS.SES` client class. To call the `sendBulkTemplatedEmail` method, create a promise for invoking an Amazon SES service object, passing the parameters. Then handle the response in the promise callback.

```
// Load the AWS SDK for Node.js  
var AWS = require("aws-sdk");
```

```
// Set the region
AWS.config.update({ region: "REGION" });

// Create sendBulkTemplatedEmail params
var params = {
  Destinations: [
    /* required */
    {
      Destination: {
        /* required */
        CcAddresses: [
          "EMAIL_ADDRESS",
          /* more items */
        ],
        ToAddresses: [
          "EMAIL_ADDRESS",
          "EMAIL_ADDRESS",
          /* more items */
        ],
      },
      ReplacementTemplateData: '{ "REPLACEMENT_TAG_NAME":"REPLACEMENT_VALUE" }',
    },
  ],
  Source: "EMAIL_ADDRESS" /* required */,
  Template: "TEMPLATE_NAME" /* required */,
  DefaultTemplateData: '{ "REPLACEMENT_TAG_NAME":"REPLACEMENT_VALUE" }',
  ReplyToAddresses: ["EMAIL_ADDRESS"],
};

// Create the promise and SES service object
var sendPromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .sendBulkTemplatedEmail(params)
  .promise();

// Handle promise's fulfilled/rejected states
sendPromise
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.log(err, err.stack);
  });
```

To run the example, type the following at the command line. The email is queued for sending by Amazon SES.

```
node ses_sendbulktemplatedemail.js
```

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

Using IP Address Filters for Email Receipt in Amazon SES



This Node.js code example shows:

- Create IP address filters to accept or reject mail that originates from an IP address or range of IP addresses.
- List your current IP address filters.
- Delete an IP address filter.

In Amazon SES, a *filter* is a data structure that consists of a name, an IP address range, and whether 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 `AWS.SES` client class:

- [`createReceiptFilter`](#)
- [`listReceiptFilters`](#)
- [`deleteReceiptFilter`](#)

Prerequisite Tasks

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

- Install Node.js. For more information about installing Node.js, see the [Node.js website](#).
- 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](#).

Configuring the SDK

Configure the SDK for JavaScript by creating a global configuration object then setting the Region for your code. In this example, the Region is set to `us-west-2`.

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

Creating an IP Address Filter

In this example, use a Node.js module to send email with Amazon SES. Create a Node.js module with the file name `ses_createreceiptfilter.js`. Configure the SDK as previously shown.

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 `createReceiptFilter` method, create a promise for invoking an Amazon SES service object, passing the parameters. Then handle the response in the promise callback.

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

// Create createReceiptFilter params
var params = {
  Filter: {
    IpFilter: {
      Cidr: "IP_ADDRESS_OR_RANGE",
```



```
    Policy: "Allow" | "Block",
  },
  Name: "NAME",
},
};

// Create the promise and SES service object
var sendPromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .createReceiptFilter(params)
  .promise();

// Handle promise's fulfilled/rejected states
sendPromise
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

To run the example, type the following at the command line. The filter is created in Amazon SES.

```
node ses_createreceiptfilter.js
```

This sample 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 Node.js module with the file name `ses_listreceiptfilters.js`. Configure the SDK as previously shown.

Create an empty parameters object. To call the `listReceiptFilters` method, create a promise for invoking an Amazon SES service object, passing the parameters. Then handle the response in the promise callback.

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

// Create the promise and SES service object
var sendPromise = new AWS.SES({ apiVersion: "2010-12-01" })
```

```
.listReceiptFilters({})
.promise();

// Handle promise's fulfilled/rejected states
sendPromise
  .then(function (data) {
    console.log(data.Filters);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

To run the example, type the following at the command line. Amazon SES returns the filter list.

```
node ses_listreceiptfilters.js
```

This sample 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 Node.js module with the file name `ses_deletereceiptfilter.js`. Configure the SDK as previously shown.

Create an object to pass the name of the IP filter to delete. To call the `deleteReceiptFilter` method, create a promise for invoking an Amazon SES service object, passing the parameters. Then handle the response in the promise callback.

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

// Create the promise and SES service object
var sendPromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .deleteReceiptFilter({ FilterName: "NAME" })
  .promise();

// Handle promise's fulfilled/rejected states
sendPromise
  .then(function (data) {
    console.log("IP Filter deleted");
  })
```

```
.catch(function (err) {  
    console.error(err, err.stack);  
});
```

To run the example, type the following at the command line. The filter is deleted from Amazon SES.

```
node ses_deletereceiptfilter.js
```

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

Using Receipt Rules in Amazon SES



This Node.js code example shows:

- Create and delete receipt rules.
- 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 AWS.SES client class:

- [createReceiptRule](#)
- [deleteReceiptRule](#)
- [createReceiptRuleSet](#)
- [deleteReceiptRuleSet](#)

Prerequisite Tasks

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

- Install Node.js. For more information about installing Node.js, see the [Node.js website](#).
- 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](#).

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 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 `createReceiptRuleSet` method, create a promise for invoking an Amazon SES service object, passing the parameters. Then handle the response in the promise callback.

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

```
// Create createReceiptRule params
var params = {
  Rule: {
    Actions: [
      {
        S3Action: {
          BucketName: "S3_BUCKET_NAME",
          ObjectKeyPrefix: "email",
        },
      },
    ],
    Recipients: [
      "DOMAIN | EMAIL_ADDRESS",
      /* more items */
    ],
    Enabled: true | false,
    Name: "RULE_NAME",
    ScanEnabled: true | false,
    TlsPolicy: "Optional",
  },
  RuleSetName: "RULE_SET_NAME",
};

// Create the promise and SES service object
var newRulePromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .createReceiptRule(params)
  .promise();

// Handle promise's fulfilled/rejected states
newRulePromise
  .then(function (data) {
    console.log("Rule created");
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

To run the example, type the following at the command line. Amazon SES creates the receipt rule.

```
node ses_createreceiptrule.js
```

This sample 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.

Create a parameters object to pass the name for the receipt rule to delete. To call the `deleteReceiptRule` method, create a promise for invoking an Amazon SES service object, passing the parameters. Then handle the response in the promise callback.

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

// Create deleteReceiptRule params
var params = {
  RuleName: "RULE_NAME" /* required */,
  RuleSetName: "RULE_SET_NAME" /* required */,
};

// Create the promise and SES service object
var newRulePromise = new AWS.SES({ apiVersion: "2010-12-01" })
  .deleteReceiptRule(params)
  .promise();

// Handle promise's fulfilled/rejected states
newRulePromise
  .then(function (data) {
    console.log("Receipt Rule Deleted");
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

To run the example, type the following at the command line. Amazon SES creates the receipt rule set list.

```
node ses_deletereceiptrule.js
```

This sample 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.

Create a parameters object to pass the name for the new receipt rule set. To call the `createReceiptRuleSet` method, create a promise for invoking an Amazon SES service object, passing the parameters. Then handle the response in the promise callback.

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

// Create the promise and SES service object
var newRulePromise = new AWS.SES({ apiVersion: "2010-12-01" })
    .createReceiptRuleSet({ RuleSetName: "NAME" })
    .promise();

// Handle promise's fulfilled/rejected states
newRulePromise
    .then(function (data) {
        console.log(data);
    })
    .catch(function (err) {
        console.error(err, err.stack);
    });
```

To run the example, type the following at the command line. Amazon SES creates the receipt rule set list.

```
node ses_createreceiptruleset.js
```

This sample 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.

Create an object to pass the name for the receipt rule set to delete. To call the `deleteReceiptRuleSet` method, create a promise for invoking an Amazon SES service object, passing the parameters. Then handle the response in the promise callback.

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

// Create the promise and SES service object
var newRulePromise = new AWS.SES({ apiVersion: "2010-12-01" })
    .deleteReceiptRuleSet({ RuleSetName: "NAME" })
    .promise();

// Handle promise's fulfilled/rejected states
newRulePromise
    .then(function (data) {
        console.log(data);
    })
    .catch(function (err) {
        console.error(err, err.stack);
    });
```

To run the example, type the following at the command line. Amazon SES creates the receipt rule set list.

```
node ses_deletereceiptruleset.js
```

This sample 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, 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: `AWS.SNS`](#).

Topics

- [Managing Topics in Amazon SNS](#)
- [Publishing Messages in Amazon SNS](#)
- [Managing Subscriptions in Amazon SNS](#)
- [Sending SMS Messages with Amazon SNS](#)

Managing Topics in Amazon SNS



This Node.js code example shows:

- How to create topics in Amazon SNS to which you can publish notifications.
- How to delete topics created in Amazon SNS.
- How to get a list of available topics.

- How to get and set topic attributes.

The Scenario

In this example, you use a series of Node.js modules to create, list, and delete Amazon SNS topics, and to handle topic attributes. The Node.js modules use the SDK for JavaScript to manage topics using these methods of the `AWS.SNS` client class:

- [createTopic](#)
- [listTopics](#)
- [deleteTopic](#)
- [getTopicAttributes](#)
- [setTopicAttributes](#)

Prerequisite Tasks

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

- Install Node.js. For more information about installing Node.js, see the [Node.js website](#).
- 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](#).

Creating a Topic

In this example, use a Node.js module to create an Amazon SNS topic. Create a Node.js module with the file name `sns_createtopic.js`. Configure the SDK as previously shown.

Create an object to pass the Name for the new topic to the `createTopic` method of the `AWS.SNS` client class. To call the `createTopic` method, create a promise for invoking an Amazon SNS service object, passing the parameters object. Then handle the response in the promise callback. The data returned by the promise contains the ARN of the topic.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set region
```

```
AWS.config.update({ region: "REGION" });

// Create promise and SNS service object
var createTopicPromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .createTopic({ Name: "TOPIC_NAME" })
  .promise();

// Handle promise's fulfilled/rejected states
createTopicPromise
  .then(function (data) {
    console.log("Topic ARN is " + data.TopicArn);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

To run the example, type the following at the command line.

```
node sns_createtopic.js
```

This sample 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 Node.js module with the file name `sns_listtopics.js`. Configure the SDK as previously shown.

Create an empty object to pass to the `listTopics` method of the `AWS.SNS` client class. To call the `listTopics` method, create a promise for invoking an Amazon SNS service object, passing the parameters object. Then handle the response in the promise callback. The data returned by the promise contains an array of your topic ARNs.

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

// Create promise and SNS service object
var listTopicsPromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .listTopics({})
  .promise();
```

```
// Handle promise's fulfilled/rejected states
listTopicsPromise
  .then(function (data) {
    console.log(data.Topics);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

To run the example, type the following at the command line.

```
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 Node.js module with the file name `sns_deletetopic.js`. Configure the SDK as previously shown.

Create an object containing the `TopicArn` of the topic to delete to pass to the `deleteTopic` method of the `AWS.SNS` client class. To call the `deleteTopic` method, create a promise for invoking an Amazon SNS service object, passing the parameters object. Then handle the response in the promise callback.

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

// Create promise and SNS service object
var deleteTopicPromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .deleteTopic({ TopicArn: "TOPIC_ARN" })
  .promise();

// Handle promise's fulfilled/rejected states
deleteTopicPromise
  .then(function (data) {
    console.log("Topic Deleted");
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

```
});
```

To run the example, type the following at the command line.

```
node sns_deletetopic.js
```

This sample 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 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 `getTopicAttributes` method of the `AWS.SNS` client class. To call the `getTopicAttributes` method, create a promise for invoking an Amazon SNS service object, passing the parameters object. Then handle the response in the promise callback.

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

// Create promise and SNS service object
var getTopicAttribsPromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .getTopicAttributes({ TopicArn: "TOPIC_ARN" })
  .promise();

// Handle promise's fulfilled/rejected states
getTopicAttribsPromise
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

To run the example, type the following at the command line.

```
node sns_gettopicattributes.js
```

This sample 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 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 `setTopicAttributes` method of the `AWS.SNS` client class. To call the `setTopicAttributes` method, create a promise for invoking an Amazon SNS service object, passing the parameters object. Then handle the response in the promise callback.

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

// Create setTopicAttributes parameters
var params = {
  AttributeName: "ATTRIBUTE_NAME" /* required */,
  TopicArn: "TOPIC_ARN" /* required */,
  AttributeValue: "NEW_ATTRIBUTE_VALUE",
};

// Create promise and SNS service object
var setTopicAttribsPromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .setTopicAttributes(params)
  .promise();

// Handle promise's fulfilled/rejected states
setTopicAttribsPromise
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

To run the example, type the following at the command line.

```
node sns_settopicattributes.js
```

This sample 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 AWS .SNS client class:

- [publish](#)

Prerequisite Tasks

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

- Install Node.js. For more information about installing Node.js, see the [Node.js website](#).
- 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](#).

Publishing a Message to an Amazon SNS Topic

In this example, use a Node.js module to publish a message to an Amazon SNS topic. Create a Node.js module with the file name `sns_publish_totopic.js`. Configure the SDK as previously shown.

Create an object containing the parameters for publishing a message, including the message text and the ARN of the Amazon SNS topic. For details on available SMS attributes, see [SetSMSAttributes](#).

Pass the parameters to the `publish` method of the `AWS.SNS` client class. Create a promise for invoking an Amazon SNS service object, passing the parameters object. Then handle the response in the promise callback.

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

// Create publish parameters
var params = {
  Message: "MESSAGE_TEXT" /* required */,
  TopicArn: "TOPIC_ARN",
};

// Create promise and SNS service object
var publishTextPromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .publish(params)
  .promise();

// Handle promise's fulfilled/rejected states
publishTextPromise
  .then(function (data) {
    console.log(
      `Message ${params.Message} sent to the topic ${params.TopicArn}`
    );
    console.log("MessageID is " + data.MessageId);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

To run the example, type the following at the command line.

```
node sns_publishtotopic.js
```

This sample 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 `AWS.SNS` client class:

- [`subscribe`](#)
- [`confirmSubscription`](#)
- [`listSubscriptionsByTopic`](#)
- [`unsubscribe`](#)

Prerequisite Tasks

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

- Install Node.js. For more information about installing Node.js, see the [Node.js website](#).
- 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](#).

Listing Subscriptions to a Topic

In this example, use a Node.js module to list all subscriptions to an Amazon SNS topic. 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 `listSubscriptionsByTopic` method of the `AWS.SNS` client class. To call the `listSubscriptionsByTopic` method, create a promise for invoking an Amazon SNS service object, passing the parameters object. Then handle the response in the promise callback.

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

const params = {
  TopicArn: "TOPIC_ARN",
};

// Create promise and SNS service object
var sublistPromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .listSubscriptionsByTopic(params)
  .promise();

// Handle promise's fulfilled/rejected states
sublistPromise
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

To run the example, type the following at the command line.

```
node sns_listsubscriptions.js
```

This sample 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 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 `subscribe` method of the `AWS.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 `subscribe` method, create a promise for invoking an Amazon SNS service object, passing the parameters object. Then handle the response in the promise callback.

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

// Create subscribe/email parameters
var params = {
  Protocol: "EMAIL" /* required */,
  TopicArn: "TOPIC_ARN" /* required */,
  Endpoint: "EMAIL_ADDRESS",
};

// Create promise and SNS service object
var subscribePromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .subscribe(params)
  .promise();

// Handle promise's fulfilled/rejected states
subscribePromise
  .then(function (data) {
    console.log("Subscription ARN is " + data.SubscriptionArn);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

To run the example, type the following at the command line.

```
node sns_subscribeemail.js
```

This sample 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 Node.js module with the file name `sns_subscribeapp.js`. Configure the SDK as previously shown.

Create an object containing the `Protocol` parameter to specify the application protocol, the `TopicArn` for the topic to subscribe to, and the ARN of a mobile application endpoint for the `Endpoint` parameter. Pass the parameters to the `subscribe` method of the `AWS.SNS` client class.

To call the `subscribe` method, create a promise for invoking an Amazon SNS service object, passing the parameters object. Then handle the response in the promise callback.

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

// Create subscribe/email parameters
var params = {
  Protocol: "application" /* required */,
  TopicArn: "TOPIC_ARN" /* required */,
  Endpoint: "MOBILE_ENDPOINT_ARN",
};

// Create promise and SNS service object
var subscribePromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .subscribe(params)
  .promise();

// Handle promise's fulfilled/rejected states
subscribePromise
  .then(function (data) {
    console.log("Subscription ARN is " + data.SubscriptionArn);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

```
});
```

To run the example, type the following at the command line.

```
node sns_subscribeapp.js
```

This sample 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 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 ARN of an AWS Lambda function as the `Endpoint` parameter. Pass the parameters to the `subscribe` method of the `AWS.SNS` client class.

To call the `subscribe` method, create a promise for invoking an Amazon SNS service object, passing the parameters object. Then handle the response in the promise callback.

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

// Create subscribe/email parameters
var params = {
  Protocol: "lambda" /* required */,
  TopicArn: "TOPIC_ARN" /* required */,
  Endpoint: "LAMBDA_FUNCTION_ARN",
};

// Create promise and SNS service object
var subscribePromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .subscribe(params)
  .promise();

// Handle promise's fulfilled/rejected states
subscribePromise
  .then(function (data) {
```

```
    console.log("Subscription ARN is " + data.SubscriptionArn);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

To run the example, type the following at the command line.

```
node sns_subscribelambda.js
```

This sample 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 Node.js module with the file name `sns_unsubscribe.js`. Configure the SDK as previously shown.

Create an object containing the `SubscriptionArn` parameter, specifying the ARN of the subscription to unsubscribe. Pass the parameters to the `unsubscribe` method of the `AWS.SNS` client class.

To call the `unsubscribe` method, create a promise for invoking an Amazon SNS service object, passing the parameters object. Then handle the response in the promise callback.

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

// Create promise and SNS service object
var subscribePromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .unsubscribe({ SubscriptionArn: TOPIC_SUBSCRIPTION_ARN })
  .promise();

// Handle promise's fulfilled/rejected states
subscribePromise
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.error(err, err.stack);
```

```
});
```

To run the example, type the following at the command line.

```
node sns_unsubscribe.js
```

This sample 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 `AWS.SNS` client class:

- [getSMSAttributes](#)
- [setSMSAttributes](#)
- [checkIfPhoneNumberIsOptedOut](#)
- [listPhoneNumbersOptedOut](#)
- [publish](#)

Prerequisite Tasks

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

- Install Node.js. For more information about installing Node.js, see the [Node.js website](#).
- 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](#).

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 Node.js module with the file name `sns_getsmstype.js`. Configure the SDK as previously shown. 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 `setTopicAttributes` method of the `AWS.SNS` client class. To call the `getSMSAttributes` method, create a promise for invoking an Amazon SNS service object, passing the parameters object. Then handle the response in the promise callback.

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

// Create SMS Attribute parameter you want to get
var params = {
  attributes: [
    "DefaultSMSType",
    "ATTRIBUTE_NAME",
    /* more items */
  ]
}
```



```
    ],  
  };  
  
  // Create promise and SNS service object  
  var getSMSTypePromise = new AWS.SNS({ apiVersion: "2010-03-31" })  
    .getSMSAttributes(params)  
    .promise();  
  
  // Handle promise's fulfilled/rejected states  
  getSMSTypePromise  
    .then(function (data) {  
      console.log(data);  
    })  
    .catch(function (err) {  
      console.error(err, err.stack);  
    });  
});
```

To run the example, type the following at the command line.

```
node sns_getsmstype.js
```

This sample 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 Node.js module with the file name `sns_setsmstype.js`. Configure the SDK as previously shown. 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 `setTopicAttributes` method of the `AWS.SNS` client class. To call the `getSMSAttributes` method, create a promise for invoking an Amazon SNS service object, passing the parameters object. Then handle the response in the promise callback.

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

```
// Create SMS Attribute parameters
var params = {
  attributes: {
    /* required */
    DefaultSMSType: "Transactional" /* highest reliability */,
    //'DefaultSMSType': 'Promotional' /* lowest cost */
  },
};

// Create promise and SNS service object
var setSMSTypePromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .setSMSAttributes(params)
  .promise();

// Handle promise's fulfilled/rejected states
setSMSTypePromise
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

To run the example, type the following at the command line.

```
node sns_setsmstype.js
```

This sample 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 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 `checkIfPhoneNumberIsOptedOut` method of the `AWS.SNS` client class. To call the `checkIfPhoneNumberIsOptedOut` method, create a promise for invoking an Amazon SNS service object, passing the parameters object. Then handle the response in the promise callback.

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

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

// Create promise and SNS service object
var phonenumPromise = new AWS.SNS({ apiVersion: "2010-03-31" })
    .checkIfPhoneNumberIsOptedOut({ phoneNumber: "PHONE_NUMBER" })
    .promise();

// Handle promise's fulfilled/rejected states
phonenumPromise
    .then(function (data) {
        console.log("Phone Opt Out is " + data.isOptedOut);
    })
    .catch(function (err) {
        console.error(err, err.stack);
    });
```

To run the example, type the following at the command line.

```
node sns_checkphoneoptout.js
```

This sample 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 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 `listPhoneNumbersOptedOut` method of the `AWS.SNS` client class. To call the `listPhoneNumbersOptedOut` method, create a promise for invoking an Amazon SNS service object, passing the parameters object. Then handle the response in the promise callback.

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

// Create promise and SNS service object
var phonelistPromise = new AWS.SNS({ apiVersion: "2010-03-31" })
```

```
.listPhoneNumbersOptedOut({})  
.promise();  
  
// Handle promise's fulfilled/rejected states  
phonelistPromise  
  .then(function (data) {  
    console.log(data);  
  })  
  .catch(function (err) {  
    console.error(err, err.stack);  
  });
```

To run the example, type the following at the command line.

```
node sns_listnumbersoptedout.js
```

This sample 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 Node.js module with the file name `sns_publishsms.js`. Configure the SDK as previously shown. 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 `publish` method of the `AWS.SNS` client class. To call the `publish` method, create a promise for invoking an Amazon SNS service object, passing the parameters object. Then handle the response in the promise callback.

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

```
var params = {
  Message: "TEXT_MESSAGE" /* required */,
  PhoneNumber: "E.164_PHONE_NUMBER",
};

// Create promise and SNS service object
var publishTextPromise = new AWS.SNS({ apiVersion: "2010-03-31" })
  .publish(params)
  .promise();

// Handle promise's fulfilled/rejected states
publishTextPromise
  .then(function (data) {
    console.log("MessageID is " + data.MessageId);
  })
  .catch(function (err) {
    console.error(err, err.stack);
  });
```

To run the example, type the following at the command line.

```
node sns_publishsms.js
```

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

Amazon SQS Examples

Amazon Simple Queue Service (Amazon 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 FIFO (first-in, first-out) delivery and exactly-once processing.



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

Topics

- [Using Queues in Amazon SQS](#)
- [Sending and Receiving Messages in Amazon SQS](#)
- [Managing Visibility Timeout in Amazon SQS](#)
- [Enabling Long Polling in Amazon SQS](#)
- [Using Dead Letter Queues in Amazon SQS](#)

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 `AWS.SQS` client class:

- [listQueues](#)
- [createQueue](#)
- [getQueueUrl](#)
- [deleteQueue](#)

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:

- Install Node.js. For more information about installing Node.js, see the [Node.js website](#).
- 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](#).

Listing Your Queues

Create a Node.js module with the file name `sqs_listqueues.js`. Be sure to configure the SDK as previously shown. To access Amazon SQS, create an `AWS.SQS` service object. Create a JSON object containing the parameters needed to list your queues, which by default is an empty object. Call the `listQueues` method to retrieve the list of queues. The callback returns the URLs of all queues.

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

// Create an SQS service object
var sqs = new AWS.SQS({ apiVersion: "2012-11-05" });

var params = {};

sqs.listQueues(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.QueueUrls);
  }
});
```

To run the example, type the following at the command line.

```
node sqs_listqueues.js
```

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

Creating a Queue

Create a Node.js module with the file name `sqs_createqueue.js`. Be sure to configure the SDK as previously shown. To access Amazon SQS, create an `AWS.SQS` service object. 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 `createQueue` method. The callback returns the URL of the created queue.

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

// Create an SQS service object
var sqs = new AWS.SQS({ apiVersion: "2012-11-05" });

var params = {
  QueueName: "SQS_QUEUE_NAME",
  Attributes: {
    DelaySeconds: "60",
    MessageRetentionPeriod: "86400",
  },
};

sqs.createQueue(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.QueueUrl);
  }
});
```

To run the example, type the following at the command line.

```
node sqs_createqueue.js
```

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

Getting the URL for a Queue

Create a Node.js module with the file name `sqs_getqueueurl.js`. Be sure to configure the SDK as previously shown. To access Amazon SQS, create an `AWS.SQS` service object. 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 `getQueueUrl` method. The callback returns the URL of the specified queue.

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

// Create an SQS service object
var sqs = new AWS.SQS({ apiVersion: "2012-11-05" });

var params = {
  QueueName: "SQS_QUEUE_NAME",
};

sqs.getQueueUrl(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.QueueUrl);
  }
});
```

To run the example, type the following at the command line.

```
node sqs_getqueueurl.js
```

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

Deleting a Queue

Create a Node.js module with the file name `sqs_deletequeue.js`. Be sure to configure the SDK as previously shown. To access Amazon SQS, create an `AWS.SQS` service object. 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 `deleteQueue` method.

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

// Create an SQS service object
var sqs = new AWS.SQS({ apiVersion: "2012-11-05" });

var params = {
  QueueUrl: "SQS_QUEUE_URL",
};

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

To run the example, type the following at the command line.

```
node sqs_deletequeue.js
```

This sample 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 `AWS.SQS` client class:

- [sendMessage](#)
- [receiveMessage](#)
- [deleteMessage](#)

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:

- Install Node.js. For more information about installing Node.js, see the [Node.js website](#).
- 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](#).
- Create an Amazon SQS queue. For an example of creating a queue, see [Using Queues in Amazon SQS](#).

Sending a Message to a Queue

Create a Node.js module with the file name `sqs_sendmessage.js`. Be sure to configure the SDK as previously shown. To access Amazon SQS, create an `AWS.SQS` service object. 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 `sendMessage` method. The callback returns the unique ID of the message.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
```

```
// Set the region
AWS.config.update({ region: "REGION" });

// Create an SQS service object
var sqs = new AWS.SQS({ apiVersion: "2012-11-05" });

var params = {
  // Remove DelaySeconds parameter and value for FIFO queues
  DelaySeconds: 10,
  MessageAttributes: {
    Title: {
      DataType: "String",
      StringValue: "The Whistler",
    },
    Author: {
      DataType: "String",
      StringValue: "John Grisham",
    },
    WeeksOn: {
      DataType: "Number",
      StringValue: "6",
    },
  },
  MessageBody:
    "Information about current NY Times fiction bestseller for week of 12/11/2016.",
  // MessageDeduplicationId: "TheWhistler", // Required for FIFO queues
  // MessageGroupId: "Group1", // Required for FIFO queues
  QueueUrl: "SQS_QUEUE_URL",
};

sqs.sendMessage(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.MessageId);
  }
});
```

To run the example, type the following at the command line.

```
node sqs_sendmessage.js
```

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

Receiving and Deleting Messages from a Queue

Create a Node.js module with the file name `sqs_receivemessage.js`. Be sure to configure the SDK as previously shown. To access Amazon SQS, create an `AWS.SQS` service object. 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 `receiveMessage` 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 `deleteMessage` method to delete the message you received.

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

// Create an SQS service object
var sqs = new AWS.SQS({ apiVersion: "2012-11-05" });

var queueURL = "SQS_QUEUE_URL";

var params = {
  AttributeNames: ["SentTimestamp"],
  MaxNumberOfMessages: 10,
  MessageAttributeNames: ["All"],
  QueueUrl: queueURL,
  VisibilityTimeout: 20,
  WaitTimeSeconds: 0,
};

sqs.receiveMessage(params, function (err, data) {
  if (err) {
    console.log("Receive Error", err);
  } else if (data.Messages) {
    var deleteParams = {
      QueueUrl: queueURL,
      ReceiptHandle: data.Messages[0].ReceiptHandle,
    };
    sqs.deleteMessage(deleteParams, function (err, data) {
```

```
    if (err) {
      console.log("Delete Error", err);
    } else {
      console.log("Message Deleted", data);
    }
  });
}
```

To run the example, type the following at the command line.

```
node sqs_receivemessage.js
```

This sample 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 `AWS.SQS` client class:

- [changeMessageVisibility](#)

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:

- Install Node.js. For more information about installing Node.js, see the [Node.js website](#).
- 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](#).
- Create an Amazon SQS queue. For an example of creating a queue, see [Using Queues in Amazon SQS](#).
- Send a message to the queue. For an example of sending a message to a queue, see [Sending and Receiving Messages in Amazon SQS](#).

Changing the Visibility Timeout

Create a Node.js module with the file name `sqs_changingvisibility.js`. Be sure to configure the SDK as previously shown. To access Amazon Simple Queue Service, create an `AWS.SQS` service object. 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 `changeMessageVisibility` method.

```
// Load the AWS SDK for Node.js
var AWS = require("aws-sdk");
// Set the region to us-west-2
AWS.config.update({ region: "us-west-2" });

// Create the SQS service object
var sqs = new AWS.SQS({ apiVersion: "2012-11-05" });

var queueURL = "https://sqs.REGION.amazonaws.com/ACCOUNT-ID/QUEUE-NAME";

var params = {
  AttributeNames: ["SentTimestamp"],
  MaxNumberOfMessages: 1,
  MessageAttributeNames: ["All"],
  QueueUrl: queueURL,
};

sqs.receiveMessage(params, function (err, data) {
  if (err) {
```

```
    console.log("Receive Error", err);
  } else {
    // Make sure we have a message
    if (data.Messages != null) {
      var visibilityParams = {
        QueueUrl: queueURL,
        ReceiptHandle: data.Messages[0].ReceiptHandle,
        VisibilityTimeout: 20, // 20 second timeout
      };
      sqs.changeMessageVisibility(visibilityParams, function (err, data) {
        if (err) {
          console.log("Delete Error", err);
        } else {
          console.log("Timeout Changed", data);
        }
      });
    } else {
      console.log("No messages to change");
    }
  }
});
```

To run the example, type the following at the command line.

```
node sqs_changingvisibility.js
```

This sample 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 `AWS.SQS` client class:

- [setQueueAttributes](#)
- [receiveMessage](#)
- [createQueue](#)

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:

- Install Node.js. For more information about installing Node.js, see the [Node.js website](#).
- 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](#).

Enabling Long Polling When Creating a Queue

Create a Node.js module with the file name `sqs_longpolling_createqueue.js`. Be sure to configure the SDK as previously shown. To access Amazon SQS, create an `AWS.SQS` service object. Create a JSON object containing the parameters needed to create a queue, including a non-zero value for the `ReceiveMessageWaitTimeSeconds` parameter. Call the `createQueue` method. Long polling is then enabled for the queue.

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

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

// Create the SQS service object
var sqs = new AWS.SQS({ apiVersion: "2012-11-05" });

var params = {
  QueueName: "SQS_QUEUE_NAME",
  Attributes: {
    ReceiveMessageWaitTimeSeconds: "20",
  },
};

sqs.createQueue(params, function (err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data.QueueUrl);
  }
});
```

To run the example, type the following at the command line.

```
node sqs_longpolling_createqueue.js
```

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

Enabling Long Polling on an Existing Queue

Create a Node.js module with the file name `sqs_longpolling_existingqueue.js`. Be sure to configure the SDK as previously shown. To access Amazon Simple Queue Service, create an `AWS.SQS` service object. Create a JSON object containing the parameters needed to set the attributes of queue, including a non-zero value for the `ReceiveMessageWaitTimeSeconds` parameter and the URL of the queue. Call the `setQueueAttributes` method. Long polling is then enabled for the queue.

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

```
// Create the SQS service object
var sqs = new AWS.SQS({ apiVersion: "2012-11-05" });

var params = {
  Attributes: {
    ReceiveMessageWaitTimeSeconds: "20",
  },
  QueueUrl: "SQS_QUEUE_URL",
};

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

To run the example, type the following at the command line.

```
node sqs_longpolling_existingqueue.js
```

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

Enabling Long Polling on Message Receipt

Create a Node.js module with the file name `sqs_longpolling_receivemessage.js`. Be sure to configure the SDK as previously shown. To access Amazon Simple Queue Service, create an `AWS.SQS` service object. 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 `receiveMessage` method.

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

// Create the SQS service object
var sqs = new AWS.SQS({ apiVersion: "2012-11-05" });

var queueURL = "SQS_QUEUE_URL";
```

```
var params = {
  AttributeNames: ["SentTimestamp"],
  MaxNumberOfMessages: 1,
  MessageAttributeNames: ["All"],
  QueueUrl: queueURL,
  WaitTimeSeconds: 20,
};

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

To run the example, type the following at the command line.

```
node sqs_longpolling_receivemessage.js
```

This sample 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 `AWS.SQS` client class:

- [setQueueAttributes](#)

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:

- Install Node.js. For more information about installing Node.js, see the [Node.js website](#).
- 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](#).
- 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](#).

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 Node.js module with the file name `sqs_deadletterqueue.js`. Be sure to configure the SDK as previously shown. To access Amazon SQS, create an `AWS.SQS` service object. 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 `setQueueAttributes` method.

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

```
// Create the SQS service object
var sqs = new AWS.SQS({ apiVersion: "2012-11-05" });

var params = {
  Attributes: {
    RedrivePolicy:
      '{"deadLetterTargetArn":"DEAD_LETTER_QUEUE_ARN","maxReceiveCount":"10"}',
  },
  QueueUrl: "SOURCE_QUEUE_URL",
};

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

To run the example, type the following at the command line.

```
node sqs_deadletterqueue.js
```

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

Tutorials

The following tutorials show you how to perform different tasks related to using the AWS SDK for JavaScript.

Topics

- [Tutorial: Setting Up Node.js on an Amazon EC2 Instance](#)

Tutorial: Setting Up Node.js on an Amazon EC2 Instance

A common scenario for using Node.js with the SDK for JavaScript is to set up and run a Node.js web application on an Amazon Elastic Compute Cloud (Amazon EC2) instance. In this tutorial, you will create a Linux instance, connect to it using SSH, and then install Node.js to run on that instance.

Prerequisites

This tutorial assumes that you have already launched a Linux instance with a public DNS name that is reachable from the Internet and to which you are able to connect using SSH. For more information, see [Step 1: Launch an Instance](#) in the *Amazon EC2 User Guide*.

Important

Use the **Amazon Linux 2023** Amazon Machine Image (AMI) when launching a new Amazon EC2 instance.

You must also have configured your security group to allow SSH (port 22), HTTP (port 80), and HTTPS (port 443) connections. For more information about these prerequisites, see [Setting Up with Amazon Amazon EC2](#) in the *Amazon EC2 User Guide*.

Procedure

The following procedure helps you install Node.js on an Amazon Linux instance. You can use this server to host a Node.js web application.

To set up Node.js on your Linux instance

1. Connect to your Linux instance as `ec2-user` using SSH.

2. Install node version manager (nvm) by typing the following at the command line.

 Warning

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

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.7/install.sh | bash
```

We will use nvm to install Node.js because nvm can install multiple versions of Node.js and allow you to switch between them.

3. Load nvm by typing the following at the command line.

```
source ~/.bashrc
```

4. Use nvm to install the latest LTS version of Node.js by typing the following at the command line.

```
nvm install --lts
```

Installing Node.js also installs the Node Package Manager (npm), so you can install additional modules as needed.

5. Test that Node.js is installed and running correctly by typing the following at the command line.

```
node -e "console.log('Running Node.js ' + process.version)"
```

This displays the following message that shows the version of Node.js that is running.

Running Node.js *VERSION*

 Note

The node installation only applies to the current Amazon EC2 session. If you restart your CLI session you need to use nvm to enable the installed node version. If the instance is

terminated, you need to install node again. The alternative is to make an Amazon Machine Image (AMI) of the Amazon EC2 instance once you have the configuration that you want to keep, as described in the following topic.

Creating an Amazon Machine Image

After you install Node.js on an Amazon EC2 instance, you can create an Amazon Machine Image (AMI) from that instance. Creating an AMI makes it easy to provision multiple Amazon EC2 instances with the same Node.js installation. For more information about creating an AMI from an existing instance, see [Creating an Amazon EBS-Backed Linux AMI](#) in the *Amazon EC2 User Guide*.

Related Resources

For more information about the commands and software used in this topic, see the following web pages:

- node version manager (nvm): see [nvm repo on GitHub](#).
- node package manager (npm): see [npm website](#).

JavaScript API Reference

The API Reference topics for the latest version of the SDK for JavaScript are found at:

[AWS SDK for JavaScript API Reference Guide.](#)

SDK Changelog on GitHub

The changelog for releases from version 2.4.8 and later is found at:

[Change log.](#)

Migrate to v3 of the AWS SDK for JavaScript

The AWS SDK for JavaScript version 3 is a major rewrite of version 2. For more information on migrating to version 3, see [Migrate from version 2.x to 3.x of the AWS SDK for JavaScript](#) in the *AWS SDK for JavaScript Developer Guide v3*.

Security for this AWS Product or Service

Cloud security at Amazon Web Services (AWS) is the highest priority. As an AWS customer, you benefit from a data center and network architecture that is built to meet the requirements of the most security-sensitive organizations. Security is a shared responsibility between AWS and you. The [Shared Responsibility Model](#) describes this as Security of the Cloud and Security in the Cloud.

Security of the Cloud – AWS is responsible for protecting the infrastructure that runs all of the services offered in the AWS Cloud and providing you with services that you can use securely. Our security responsibility is the highest priority at AWS, and the effectiveness of our security is regularly tested and verified by third-party auditors as part of the [AWS Compliance Programs](#).

Security in the Cloud – Your responsibility is determined by the AWS service you are using, and other factors including the sensitivity of your data, your organization's requirements, and applicable laws and regulations.

This AWS product or service follows the [shared responsibility model](#) through the specific Amazon Web Services (AWS) services it supports. For AWS service security information, see the [AWS service security documentation page](#) and [AWS services that are in scope of AWS compliance efforts by compliance program](#).

Topics

- [Data protection in this AWS product or service](#)
- [Identity and Access Management](#)
- [Compliance Validation for this AWS Product or Service](#)
- [Resilience for this AWS Product or Service](#)
- [Infrastructure Security for this AWS Product or Service](#)
- [Enforcing a minimum version of TLS](#)

Data protection in this AWS product or service

The AWS [shared responsibility model](#) applies to data protection in this AWS product or service. As described in this model, AWS is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this infrastructure. You are also responsible for the security configuration and management tasks

for the AWS services that you use. For more information about data privacy, see the [Data Privacy FAQ](#). For information about data protection in Europe, see the [AWS Shared Responsibility Model and GDPR](#) blog post on the *AWS Security Blog*.

For data protection purposes, we recommend that you protect AWS account credentials and set up individual users with AWS IAM Identity Center or AWS Identity and Access Management (IAM). That way, each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources. We require TLS 1.2 and recommend TLS 1.3.
- Set up API and user activity logging with AWS CloudTrail. For information about using CloudTrail trails to capture AWS activities, see [Working with CloudTrail trails](#) in the *AWS CloudTrail User Guide*.
- Use AWS encryption solutions, along with all default security controls within AWS services.
- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing sensitive data that is stored in Amazon S3.
- If you require FIPS 140-3 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see [Federal Information Processing Standard \(FIPS\) 140-3](#).

We strongly recommend that you never put confidential or sensitive information, such as your customers' email addresses, into tags or free-form text fields such as a **Name** field. This includes when you work with this AWS product or service or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into tags or free-form text fields used for names may be used for billing or diagnostic logs. If you provide a URL to an external server, we strongly recommend that you do not include credentials information in the URL to validate your request to that server.

Identity and Access Management

AWS Identity and Access Management (IAM) is an AWS service that helps an administrator securely control access to AWS resources. IAM administrators control who can be *authenticated* (signed in) and *authorized* (have permissions) to use AWS resources. IAM is an AWS service that you can use with no additional charge.

Topics

- [Audience](#)
- [Authenticating with identities](#)
- [Managing access using policies](#)
- [How AWS services work with IAM](#)
- [Troubleshooting AWS identity and access](#)

Audience

How you use AWS Identity and Access Management (IAM) differs, depending on the work that you do in AWS.

Service user – If you use AWS services to do your job, then your administrator provides you with the credentials and permissions that you need. As you use more AWS features to do your work, you might need additional permissions. Understanding how access is managed can help you request the right permissions from your administrator. If you cannot access a feature in AWS, see [Troubleshooting AWS identity and access](#) or the user guide of the AWS service you are using.

Service administrator – If you're in charge of AWS resources at your company, you probably have full access to AWS. It's your job to determine which AWS features and resources your service users should access. You must then submit requests to your IAM administrator to change the permissions of your service users. Review the information on this page to understand the basic concepts of IAM. To learn more about how your company can use IAM with AWS, see the user guide of the AWS service you are using.

IAM administrator – If you're an IAM administrator, you might want to learn details about how you can write policies to manage access to AWS. To view example AWS identity-based policies that you can use in IAM, see the user guide of the AWS service you are using.

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. You must be *authenticated* (signed in to AWS) as the AWS account root user, as an IAM user, or by assuming an IAM role.

You can sign in to AWS as a federated identity by using credentials provided through an identity source. AWS IAM Identity Center (IAM Identity Center) users, your company's single sign-on

authentication, and your Google or Facebook credentials are examples of federated identities. When you sign in as a federated identity, your administrator previously set up identity federation using IAM roles. When you access AWS by using federation, you are indirectly assuming a role.

Depending on the type of user you are, you can sign in to the AWS Management Console or the AWS access portal. For more information about signing in to AWS, see [How to sign in to your AWS account](#) in the *AWS Sign-In User Guide*.

If you access AWS programmatically, AWS provides a software development kit (SDK) and a command line interface (CLI) to cryptographically sign your requests by using your credentials. If you don't use AWS tools, you must sign requests yourself. For more information about using the recommended method to sign requests yourself, see [AWS Signature Version 4 for API requests](#) in the *IAM User Guide*.

Regardless of the authentication method that you use, you might be required to provide additional security information. For example, AWS recommends that you use multi-factor authentication (MFA) to increase the security of your account. To learn more, see [Multi-factor authentication](#) in the *AWS IAM Identity Center User Guide* and [AWS Multi-factor authentication in IAM](#) in the *IAM User Guide*.

AWS account root user

When you create an AWS account, you begin with one sign-in identity that has complete access to all AWS services and resources in the account. This identity is called the AWS account *root user* and is accessed by signing in with the email address and password that you used to create the account. We strongly recommend that you don't use the root user for your everyday tasks. Safeguard your root user credentials and use them to perform the tasks that only the root user can perform. For the complete list of tasks that require you to sign in as the root user, see [Tasks that require root user credentials](#) in the *IAM User Guide*.

Federated identity

As a best practice, require human users, including users that require administrator access, to use federation with an identity provider to access AWS services by using temporary credentials.

A *federated identity* is a user from your enterprise user directory, a web identity provider, the AWS Directory Service, the Identity Center directory, or any user that accesses AWS services by using credentials provided through an identity source. When federated identities access AWS accounts, they assume roles, and the roles provide temporary credentials.

For centralized access management, we recommend that you use AWS IAM Identity Center. You can create users and groups in IAM Identity Center, or you can connect and synchronize to a set of users and groups in your own identity source for use across all your AWS accounts and applications. For information about IAM Identity Center, see [What is IAM Identity Center?](#) in the *AWS IAM Identity Center User Guide*.

IAM users and groups

An [IAM user](#) is an identity within your AWS account that has specific permissions for a single person or application. Where possible, we recommend relying on temporary credentials instead of creating IAM users who have long-term credentials such as passwords and access keys. However, if you have specific use cases that require long-term credentials with IAM users, we recommend that you rotate access keys. For more information, see [Rotate access keys regularly for use cases that require long-term credentials](#) in the *IAM User Guide*.

An [IAM group](#) is an identity that specifies a collection of IAM users. You can't sign in as a group. You can use groups to specify permissions for multiple users at a time. Groups make permissions easier to manage for large sets of users. For example, you could have a group named *IAMAdmins* and give that group permissions to administer IAM resources.

Users are different from roles. A user is uniquely associated with one person or application, but a role is intended to be assumable by anyone who needs it. Users have permanent long-term credentials, but roles provide temporary credentials. To learn more, see [Use cases for IAM users](#) in the *IAM User Guide*.

IAM roles

An [IAM role](#) is an identity within your AWS account that has specific permissions. It is similar to an IAM user, but is not associated with a specific person. To temporarily assume an IAM role in the AWS Management Console, you can [switch from a user to an IAM role \(console\)](#). You can assume a role by calling an AWS CLI or AWS API operation or by using a custom URL. For more information about methods for using roles, see [Methods to assume a role](#) in the *IAM User Guide*.

IAM roles with temporary credentials are useful in the following situations:

- **Federated user access** – To assign permissions to a federated identity, you create a role and define permissions for the role. When a federated identity authenticates, the identity is associated with the role and is granted the permissions that are defined by the role. For information about roles for federation, see [Create a role for a third-party identity provider](#)

([federation](#)) in the *IAM User Guide*. If you use IAM Identity Center, you configure a permission set. To control what your identities can access after they authenticate, IAM Identity Center correlates the permission set to a role in IAM. For information about permissions sets, see [Permission sets](#) in the *AWS IAM Identity Center User Guide*.

- **Temporary IAM user permissions** – An IAM user or role can assume an IAM role to temporarily take on different permissions for a specific task.
- **Cross-account access** – You can use an IAM role to allow someone (a trusted principal) in a different account to access resources in your account. Roles are the primary way to grant cross-account access. However, with some AWS services, you can attach a policy directly to a resource (instead of using a role as a proxy). To learn the difference between roles and resource-based policies for cross-account access, see [Cross account resource access in IAM](#) in the *IAM User Guide*.
- **Cross-service access** – Some AWS services use features in other AWS services. For example, when you make a call in a service, it's common for that service to run applications in Amazon EC2 or store objects in Amazon S3. A service might do this using the calling principal's permissions, using a service role, or using a service-linked role.
- **Forward access sessions (FAS)** – When you use an IAM user or role to perform actions in AWS, you are considered a principal. When you use some services, you might perform an action that then initiates another action in a different service. FAS uses the permissions of the principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. FAS requests are only made when a service receives a request that requires interactions with other AWS services or resources to complete. In this case, you must have permissions to perform both actions. For policy details when making FAS requests, see [Forward access sessions](#).
- **Service role** – A service role is an [IAM role](#) that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see [Create a role to delegate permissions to an AWS service](#) in the *IAM User Guide*.
- **Service-linked role** – A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.
- **Applications running on Amazon EC2** – You can use an IAM role to manage temporary credentials for applications that are running on an EC2 instance and making AWS CLI or AWS API requests. This is preferable to storing access keys within the EC2 instance. To assign an AWS role to an EC2 instance and make it available to all of its applications, you create an instance profile

that is attached to the instance. An instance profile contains the role and enables programs that are running on the EC2 instance to get temporary credentials. For more information, see [Use an IAM role to grant permissions to applications running on Amazon EC2 instances](#) in the *IAM User Guide*.

Managing access using policies

You control access in AWS by creating policies and attaching them to AWS identities or resources. A policy is an object in AWS that, when associated with an identity or resource, defines their permissions. AWS evaluates these policies when a principal (user, root user, or role session) makes a request. Permissions in the policies determine whether the request is allowed or denied. Most policies are stored in AWS as JSON documents. For more information about the structure and contents of JSON policy documents, see [Overview of JSON policies](#) in the *IAM User Guide*.

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

By default, users and roles have no permissions. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies. The administrator can then add the IAM policies to roles, and users can assume the roles.

IAM policies define permissions for an action regardless of the method that you use to perform the operation. For example, suppose that you have a policy that allows the `iam:GetRole` action. A user with that policy can get role information from the AWS Management Console, the AWS CLI, or the AWS API.

Identity-based policies

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see [Define custom IAM permissions with customer managed policies](#) in the *IAM User Guide*.

Identity-based policies can be further categorized as *inline policies* or *managed policies*. Inline policies are embedded directly into a single user, group, or role. Managed policies are standalone policies that you can attach to multiple users, groups, and roles in your AWS account. Managed policies include AWS managed policies and customer managed policies. To learn how to choose between a managed policy or an inline policy, see [Choose between managed policies and inline policies](#) in the *IAM User Guide*.

Resource-based policies

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must [specify a principal](#) in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

Resource-based policies are inline policies that are located in that service. You can't use AWS managed policies from IAM in a resource-based policy.

Access control lists (ACLs)

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

Amazon S3, AWS WAF, and Amazon VPC are examples of services that support ACLs. To learn more about ACLs, see [Access control list \(ACL\) overview](#) in the *Amazon Simple Storage Service Developer Guide*.

Other policy types

AWS supports additional, less-common policy types. These policy types can set the maximum permissions granted to you by the more common policy types.

- **Permissions boundaries** – A permissions boundary is an advanced feature in which you set the maximum permissions that an identity-based policy can grant to an IAM entity (IAM user or role). You can set a permissions boundary for an entity. The resulting permissions are the intersection of an entity's identity-based policies and its permissions boundaries. Resource-based policies that specify the user or role in the `Principal` field are not limited by the permissions boundary. An explicit deny in any of these policies overrides the allow. For more information about permissions boundaries, see [Permissions boundaries for IAM entities](#) in the *IAM User Guide*.
- **Service control policies (SCPs)** – SCPs are JSON policies that specify the maximum permissions for an organization or organizational unit (OU) in AWS Organizations. AWS Organizations is a service for grouping and centrally managing multiple AWS accounts that your business owns. If you enable all features in an organization, then you can apply service control policies (SCPs) to

any or all of your accounts. The SCP limits permissions for entities in member accounts, including each AWS account root user. For more information about Organizations and SCPs, see [Service control policies](#) in the *AWS Organizations User Guide*.

- **Resource control policies (RCPs)** – RCPs are JSON policies that you can use to set the maximum available permissions for resources in your accounts without updating the IAM policies attached to each resource that you own. The RCP limits permissions for resources in member accounts and can impact the effective permissions for identities, including the AWS account root user, regardless of whether they belong to your organization. For more information about Organizations and RCPs, including a list of AWS services that support RCPs, see [Resource control policies \(RCPs\)](#) in the *AWS Organizations User Guide*.
- **Session policies** – Session policies are advanced policies that you pass as a parameter when you programmatically create a temporary session for a role or federated user. The resulting session's permissions are the intersection of the user or role's identity-based policies and the session policies. Permissions can also come from a resource-based policy. An explicit deny in any of these policies overrides the allow. For more information, see [Session policies](#) in the *IAM User Guide*.

Multiple policy types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see [Policy evaluation logic](#) in the *IAM User Guide*.

How AWS services work with IAM

To get a high-level view of how AWS services work with most IAM features, see [AWS services that work with IAM](#) in the *IAM User Guide*.

To learn how to use a specific AWS service with IAM, see the security section of the relevant service's User Guide.

Troubleshooting AWS identity and access

Use the following information to help you diagnose and fix common issues that you might encounter when working with AWS and IAM.

Topics

- [I am not authorized to perform an action in AWS](#)
- [I am not authorized to perform iam:PassRole](#)

- [I want to allow people outside of my AWS account to access my AWS resources](#)

I am not authorized to perform an action in AWS

If you receive an error that you're not authorized to perform an action, your policies must be updated to allow you to perform the action.

The following example error occurs when the `mateojackson` IAM user tries to use the console to view details about a fictional *my-example-widget* resource but doesn't have the fictional `awes:GetWidget` permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
awes:GetWidget on resource: my-example-widget
```

In this case, the policy for the `mateojackson` user must be updated to allow access to the *my-example-widget* resource by using the `awes:GetWidget` action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I am not authorized to perform iam:PassRole

If you receive an error that you're not authorized to perform the `iam:PassRole` action, your policies must be updated to allow you to pass a role to AWS.

Some AWS services allow you to pass an existing role to that service instead of creating a new service role or service-linked role. To do this, you must have permissions to pass the role to the service.

The following example error occurs when an IAM user named `marymajor` tries to use the console to perform an action in AWS. However, the action requires the service to have permissions that are granted by a service role. Mary does not have permissions to pass the role to the service.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
iam:PassRole
```

In this case, Mary's policies must be updated to allow her to perform the `iam:PassRole` action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I want to allow people outside of my AWS account to access my AWS resources

You can create a role that users in other accounts or people outside of your organization can use to access your resources. You can specify who is trusted to assume the role. For services that support resource-based policies or access control lists (ACLs), you can use those policies to grant people access to your resources.

To learn more, consult the following:

- To learn whether AWS supports these features, see [How AWS services work with IAM](#).
- To learn how to provide access to your resources across AWS accounts that you own, see [Providing access to an IAM user in another AWS account that you own](#) in the *IAM User Guide*.
- To learn how to provide access to your resources to third-party AWS accounts, see [Providing access to AWS accounts owned by third parties](#) in the *IAM User Guide*.
- To learn how to provide access through identity federation, see [Providing access to externally authenticated users \(identity federation\)](#) in the *IAM User Guide*.
- To learn the difference between using roles and resource-based policies for cross-account access, see [Cross account resource access in IAM](#) in the *IAM User Guide*.

Compliance Validation for this AWS Product or Service

To learn whether an AWS service is within the scope of specific compliance programs, see [AWS services in Scope by Compliance Program](#) and choose the compliance program that you are interested in. For general information, see [AWS Compliance Programs](#).

You can download third-party audit reports using AWS Artifact. For more information, see [Downloading Reports in AWS Artifact](#).

Your compliance responsibility when using AWS services is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. AWS provides the following resources to help with compliance:

- [Security Compliance & Governance](#) – These solution implementation guides discuss architectural considerations and provide steps for deploying security and compliance features.
- [HIPAA Eligible Services Reference](#) – Lists HIPAA eligible services. Not all AWS services are HIPAA eligible.

- [AWS Compliance Resources](#) – This collection of workbooks and guides might apply to your industry and location.
- [AWS Customer Compliance Guides](#) – Understand the shared responsibility model through the lens of compliance. The guides summarize the best practices for securing AWS services and map the guidance to security controls across multiple frameworks (including National Institute of Standards and Technology (NIST), Payment Card Industry Security Standards Council (PCI), and International Organization for Standardization (ISO)).
- [Evaluating Resources with Rules](#) in the *AWS Config Developer Guide* – The AWS Config service assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.
- [AWS Security Hub](#) – This AWS service provides a comprehensive view of your security state within AWS. Security Hub uses security controls to evaluate your AWS resources and to check your compliance against security industry standards and best practices. For a list of supported services and controls, see [Security Hub controls reference](#).
- [Amazon GuardDuty](#) – This AWS service detects potential threats to your AWS accounts, workloads, containers, and data by monitoring your environment for suspicious and malicious activities. GuardDuty can help you address various compliance requirements, like PCI DSS, by meeting intrusion detection requirements mandated by certain compliance frameworks.
- [AWS Audit Manager](#) – This AWS service helps you continuously audit your AWS usage to simplify how you manage risk and compliance with regulations and industry standards.

This AWS product or service follows the [shared responsibility model](#) through the specific Amazon Web Services (AWS) services it supports. For AWS service security information, see the [AWS service security documentation page](#) and [AWS services that are in scope of AWS compliance efforts by compliance program](#).

Resilience for this AWS Product or Service

The AWS global infrastructure is built around AWS Regions and Availability Zones.

AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking.

With Availability Zones, you can design and operate applications and databases that automatically fail over between zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information about AWS Regions and Availability Zones, see [AWS Global Infrastructure](#).

This AWS product or service follows the [shared responsibility model](#) through the specific Amazon Web Services (AWS) services it supports. For AWS service security information, see the [AWS service security documentation page](#) and [AWS services that are in scope of AWS compliance efforts by compliance program](#).

Infrastructure Security for this AWS Product or Service

This AWS product or service uses managed services, and therefore is protected by the AWS global network security. For information about AWS security services and how AWS protects infrastructure, see [AWS Cloud Security](#). To design your AWS environment using the best practices for infrastructure security, see [Infrastructure Protection](#) in *Security Pillar AWS Well-Architected Framework*.

You use AWS published API calls to access this AWS Product or Service through the network. Clients must support the following:

- Transport Layer Security (TLS). We require TLS 1.2 and recommend TLS 1.3.
- Cipher suites with perfect forward secrecy (PFS) such as DHE (Ephemeral Diffie-Hellman) or ECDHE (Elliptic Curve Ephemeral Diffie-Hellman). Most modern systems such as Java 7 and later support these modes.

Additionally, requests must be signed by using an access key ID and a secret access key that is associated with an IAM principal. Or you can use the [AWS Security Token Service](#) (AWS STS) to generate temporary security credentials to sign requests.

This AWS product or service follows the [shared responsibility model](#) through the specific Amazon Web Services (AWS) services it supports. For AWS service security information, see the [AWS service security documentation page](#) and [AWS services that are in scope of AWS compliance efforts by compliance program](#).

Enforcing a minimum version of TLS

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.

Important

As of June 10, 2024, we [announced](#) that TLS 1.3 is available on AWS service API endpoints across each of the AWS Regions. The AWS SDK for JavaScript v2 does not negotiate the TLS version itself. Instead, it uses the TLS version determined by Node.js, which is configurable via `https.Agent`. AWS recommends using the current Active LTS version of Node.js.

Verify and enforce TLS in Node.js

When you use the AWS SDK for JavaScript with Node.js, the underlying Node.js security layer is used to set the TLS version.

Node.js 12.0.0 and later use a minimum version of OpenSSL 1.1.1b, which supports TLS 1.3. The AWS SDK for JavaScript v2 defaults to use TLS 1.3 when available, but defaults to a lower version if required.

Verify the version of OpenSSL and TLS

To get the version of OpenSSL used by Node.js on your computer, run the following command.

```
node -p process.versions
```

The version of OpenSSL in the list is the version used by Node.js, as shown in the following example.

```
openssl: '1.1.1b'
```

To get the version of TLS used by Node.js on your computer, start the Node shell and run the following commands, in order.

```
> var tls = require("tls");  
> var tlsSocket = new tls.TLSSocket();  
> tlsSocket.getProtocol();
```

The last command outputs the TLS version, as shown in the following example.

```
'TLSv1.3'
```

Node.js defaults to use this version of TLS, and tries to negotiate another version of TLS if a call is not successful.

Checking Minimum and Maximum Supported TLS Versions

Developers can check the minimum and maximum supported TLS versions in Node.js using the following script:

```
var tls = require("tls");
console.log("Supported TLS versions:", tls.DEFAULT_MIN_VERSION + " to " +
  tls.DEFAULT_MAX_VERSION);
```

The last command outputs the default minimum and maximum TLS version, as shown in the following example.

```
Supported TLS versions: TLSv1.2 to TLSv1.3
```

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 11.4.0 or later. To install a specific Node.js version, first install Node Version Manager (nvm) using the steps found at [Node Version Manager Installing and Updating](#). Then run the following commands to install and use a specific version of Node.js.

```
nvm install 11
nvm use 11
```

Enforcing TLS 1.2

To enforce that TLS 1.2 is the minimum allowable version, specify the `--tls-min-v1.2` argument when running your script, as shown in the following example.

```
node --tls-min-v1.2 yourScript.js
```

To specify the minimum allowable TLS version for a specific request in your JavaScript code, use the `httpOptions` parameter to specify the protocol, as shown in the following example.

```
const https = require("https");
const {NodeHttpHandler} = require("@aws-sdk/node-http-handler");
const {DynamoDBClient} = require("@aws-sdk/client-dynamodb");

const client = new DynamoDBClient({
  region: "us-west-2",
  requestHandler: new NodeHttpHandler({
    httpsAgent: new https.Agent({
      {
        secureProtocol: 'TLSv1_2_method'
      }
    })
  })
});
```

Enforcing TLS 1.3

To enforce that TLS 1.3 is the minimum allowable version, specify the `--tls-min-v1.3` argument when running your script, as shown in the following example.

```
node --tls-min-v1.3 yourScript.js
```

To specify the minimum allowable TLS version for a specific request in your JavaScript code, use the `httpOptions` parameter to specify the protocol, as shown in the following example.

```
const https = require("https");
const {NodeHttpHandler} = require("@aws-sdk/node-http-handler");
const {DynamoDBClient} = require("@aws-sdk/client-dynamodb");

const client = new DynamoDBClient({
  region: "us-west-2",
  requestHandler: new NodeHttpHandler({
    httpsAgent: new https.Agent({
      {
        secureProtocol: 'TLSv1_3_method'
      }
    })
  })
});
```

Verify and enforce TLS in a browser script

When you use the SDK for JavaScript in a browser script, browser settings control the version of TLS that is used. The version of TLS used by the browser cannot be discovered or set by script and must be configured by the user. To verify and enforce the version of TLS used in a browser script, refer to the instructions for your specific browser.

Microsoft Internet Explorer

1. Open **Internet Explorer**.
2. From the menu bar, choose **Tools - Internet Options - Advanced** tab.
3. Scroll down to **Security** category, manually check the option box for **Use TLS 1.2**.
4. Click **OK**.
5. Close your browser and restart Internet Explorer.

Microsoft Edge

1. In the Windows menu search box, type *Internet options*.
2. Under **Best match**, click **Internet Options**.
3. In the **Internet Properties** window, on the **Advanced** tab, scroll down to the **Security** section.
4. Check the **User TLS 1.2** checkbox.
5. Click **OK**.

Google Chrome

1. Open **Google Chrome**.
2. Click **Alt F** and select **Settings**.
3. Scroll down and select **Show advanced settings....**
4. Scroll down to the **System** section and click on **Open proxy settings....**
5. Select the **Advanced** tab.
6. Scroll down to **Security** category, manually check the option box for **Use TLS 1.2**.
7. Click **OK**.
8. Close your browser and restart Google Chrome.

Mozilla Firefox

1. Open **Firefox**.
2. In the address bar, type **about:config** and press Enter.
3. In the **Search** field, enter **tls**. Find and double-click the entry for **security.tls.version.min**.
4. Set the integer value to 3 to force protocol of TLS 1.2 to be the default.
5. Click **OK**.
6. Close your browser and restart Mozilla Firefox.

Apple Safari

There are no options for enabling SSL protocols. If you are using Safari version 7 or greater, TLS 1.2 is automatically enabled.

Additional Resources

The following links provide additional resources you can use with the [AWS SDK for JavaScript](#).

AWS SDKs and Tools Reference Guide

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

JavaScript SDK Forum

You can find questions and discussions on matters of interest to users of the SDK for JavaScript in the [JavaScript SDK Forum](#).

JavaScript SDK and Developer Guide on GitHub

There are several repositories on GitHub for the SDK for JavaScript.

- The current SDK for JavaScript is available in the [SDK repo](#).
- The SDK for JavaScript Developer Guide (this document) is available in markdown format in its own [documentation repo](#).
- Some of the sample code that is included in this guide is available in the [SDK sample code repo](#).

JavaScript SDK on Gitter

You can also find questions and discussions about the SDK for JavaScript in the [JavaScript SDK community](#) on Gitter.

Document History for AWS SDK for JavaScript

- **SDK version:** See [JavaScript API Reference](#)
- **Latest major documentation update:** March 31, 2022

Document History

The following table describes important changes in each release of the AWS SDK for JavaScript after May 2018. For notification about updates to this documentation, you can subscribe to an [RSS feed](#).

Change	Description	Date
TLS 1.3 is now supported across all AWS service API endpoints in all Regions	Updated supported TLS version and method for logging TLS version.	April 10, 2025
Enforcing a minimum version of TLS	Added information about TLS 1.3.	March 31, 2022
Viewing Photos in an Amazon S3 Bucket from a Browser	Added an example for simply viewing photos in existing photo albums.	May 13, 2019
Setting Credentials in Node.js, new credential-loading choices	Added information about credentials that are loaded from the ECS credentials provider or a configured credential process.	April 25, 2019
Credentials using a Configured Credential Process	Added information about credentials that are loaded from a configured credential process.	April 25, 2019
New Getting Started in a Browser Script	Getting Started in a Browser Script has been rewritten to	July 14, 2018

simplify the example and to access the Amazon Polly service to send text and return synthesized speech you can play in the browser. See [Getting Started in Browser Script](#) for the new content.

[New Amazon SNS Code Samples](#)

Four new Node.js code samples for working with Amazon SNS have been added. See [Amazon SNS Examples](#) for the sample code.

June 29, 2018

[New Getting Started in Node.js](#)

Getting Started in Node.js has been rewritten to use updated sample code and to provide greater detail in how to create the `package.json` file as well as the Node.js code itself. See [Getting Started in Node.js](#) for the new content.

June 4, 2018

Earlier Updates

The following table describes important changes in each release of the AWS SDK for JavaScript before June 2018.

Change	Description	Date
New AWS Elemental MediaConvert code samples	Three new Node.js code samples for working with AWS Elemental MediaConvert have been added. See AWS Elemental MediaConv	May 21, 2018

Change	Description	Date
	ert Examples for the sample code.	
New Edit on GitHub Button	The header of every topic now provides a button that takes you to the markdown version of same topic on GitHub so you can provide edits to improve the accuracy and completeness of the guide.	February 21, 2018
New Topic on Custom Endpoints	Information has been added on the format and use of custom endpoints for making API calls. See Specifying Custom Endpoints .	February 20, 2018
SDK for JavaScript Developer Guide on GitHub	The SDK for JavaScript Developer Guide is available in markdown format in its own documentation repo . You can post issues you would like the guide to address or submit pull requests to submit proposed changes.	February 16, 2018
New Amazon DynamoDB code sample	A new Node.js code sample for updating a DynamoDB table using the Document Client has been added. See Using the DynamoDB Document Client for the sample code.	February 14, 2018

Change	Description	Date
New Topic on SDK Logging	A topic describing how to log API calls made with the SDK for JavaScript has been added, including information about using a third-party logger. See Logging AWS SDK for JavaScript Calls .	February 5, 2018
Updated Topic on Region Setting	The topic describing how to set the Region used with the SDK has been updated and expanded, including information about the order of precedence for setting the Region. See Setting the AWS Region .	December 12, 2017
New Amazon SES Code Examples	The section with SDK code examples has been updated to include five new examples for working with Amazon SES. For more information about these code examples, see Amazon Simple Email Service Examples .	November 9, 2017

Change	Description	Date
Usability Improvements	Based on recent usability testing, a number of changes have been made to improve documentation usability. • Code samples are more clearly identified as targeted either for browser or Node.js execution. • TOC links no longer jump immediately to other web content, including the API Reference. • Includes more linking in Getting Started section to details on obtaining AWS credentials. • Provides more information about common Node.js features needed to use the SDK. For more information, see Node.js Considerations.	August 9, 2017
New DynamoDB Code Examples	The section with SDK code examples has been updated to re-write the two previous examples as well as add three brand new examples for working with DynamoDB. For more information about these code examples, see Amazon DynamoDB Examples.	June 21, 2017

Change	Description	Date
New IAM Code Examples	The section with SDK code examples has been updated to include five new examples for working with IAM. For more information about these code examples, see AWS IAM Examples .	December 23, 2016
New CloudWatch and Amazon SQS Code Examples	The section with SDK code examples has been updated to include new examples for working with CloudWatch and with Amazon SQS. For more information about these code examples, see Amazon CloudWatch Examples and Amazon SQS Examples .	December 20, 2016
New Amazon EC2 Code Examples	The section with SDK code examples has been updated to include five new examples for working with Amazon EC2. For more information about these code examples, see Amazon EC2 Examples .	December 15, 2016
List of supported browsers made more visible	The list of browsers supported by the SDK for JavaScript, which was previously found in the topic on Prerequisites, has been given its own topic to make it more visible in the table of contents.	November 16, 2016

Change	Description	Date
Initial publication of the new Developer Guide	The previous Developer Guide is now deprecated. The new Developer Guide has been reorganized to make information easier to find. When either Node.js or browser JavaScript scenarios present special considerations, those are identified as appropriate. The guide also provides additional code examples that are better organized to make them easier and faster to find.	October 28, 2016