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

A common scenario for using Node.js with the SDK for JavaScript is to set up and run a Node.js web application on an Amazon Elastic Compute Cloud (Amazon EC2) instance. In this tutorial, you will create a Linux instance, connect to it using SSH, and then install Node.js to run on that instance.

Prerequisites

This tutorial assumes that you have already launched a Linux instance with a public DNS name that is reachable from the Internet and to which you are able to connect using SSH. For more information, see Step 1: Launch an Instance in the *Amazon EC2 User Guide for Linux Instances*.

You must also have configured your security group to allow `SSH` (port 22), `HTTP` (port 80), and `HTTPS` (port 443) connections. For more information about these prerequisites, see Setting Up with Amazon EC2 in the *Amazon EC2 User Guide for Linux Instances*.

Procedure

The following procedure helps you install Node.js on an Amazon Linux instance. You can use this server to host a Node.js web application.

To set up Node.js on your Linux instance

1. Connect to your Linux instance as `ec2-user` using SSH.
2. Install node version manager (nvm) by typing the following at the command line.

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

   ```
   curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.34.0/install.sh | bash
   ```

   We will use nvm to install Node.js because nvm can install multiple versions of Node.js and allow you to switch between them.
3. Activate nvm by typing the following at the command line.

   ```
   . ~/.nvm/nvm.sh
   ```

4. Use nvm to install the latest version of Node.js by typing the following at the command line.

```
nvm install node
```

   Installing Node.js also installs the Node Package Manager (npm) so you can install additional modules as needed.

5. Test that Node.js is installed and running correctly by typing the following at the command line.

```
node -e "console.log('Running Node.js ' + process.version)"
```

   This displays the following message that shows the version of Node.js that is running.

```
Running Node.js VERSION
```

   Note
   The node installation only applies to the current EC2 session. Once the EC2 instance goes away, you'll have to re-install node again. The alternative is to make an AMI of the EC2 instance once you have the configuration that you want to keep, as described in the following section.

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 for Linux Instances*.

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.

Tutorial: Creating and Using Lambda Functions

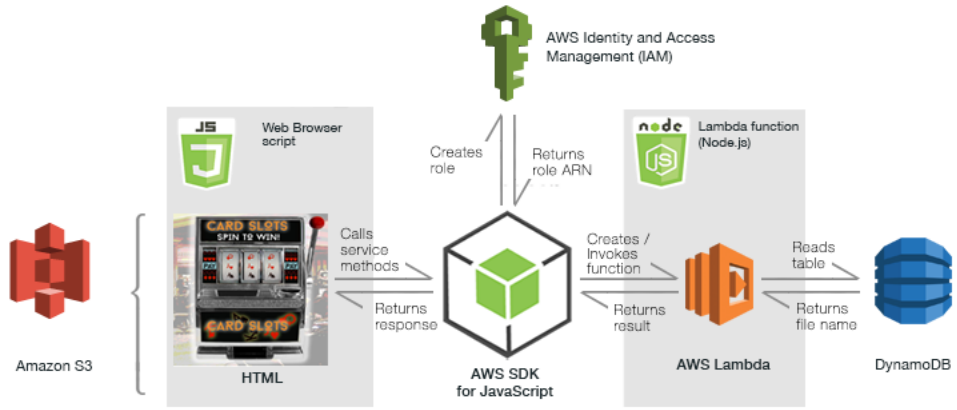
In this tutorial, you learn how to:

- Create AWS Lambda functions in Node.js and call them from JavaScript running in a web browser.
- Call another service within a Lambda function and process the asynchronous responses before forwarding those responses to the browser script.
- Use Node.js scripts to create the resources needed by the Lambda function.

The Scenario

In this example, a simulated browser-based slot machine game invokes a Lambda function that generates the random results of each slot pull. Those results are returned as the file names of the images that are used to display to the user. The images are stored in an Amazon S3 bucket that is configured to function as a static web host for the HTML, CSS, and other assets used to present the application experience.

This diagram illustrates most of the elements in this application and how they relate to one another. Versions of this diagram will appear throughout the tutorial to show the focus of each task



Prerequisites

You must complete the following tasks before you can begin the tutorial:

- Install Node.js on your computer to run various scripts that help set up the Amazon S3 bucket and the Amazon DynamoDB table, and create and configure the Lambda function. The Lambda function itself runs in the AWS Lambda Node.js environment. For information about installing Node.js, see www.nodejs.org.
- Install the AWS SDK for JavaScript on your computer to run the setup scripts. For information on installing the AWS SDK for JavaScript for Node.js, see Installing the SDK for JavaScript (p. 16).

The tutorial should take about 30 minutes to complete.

Tutorial Steps

To create this application you'll need resources from multiple services that must be connected and configured in both the code of the browser script and the Node.js code of the Lambda function.

To construct the tutorial application and the Lambda function it uses

1. Create a working directory on your computer for this tutorial.

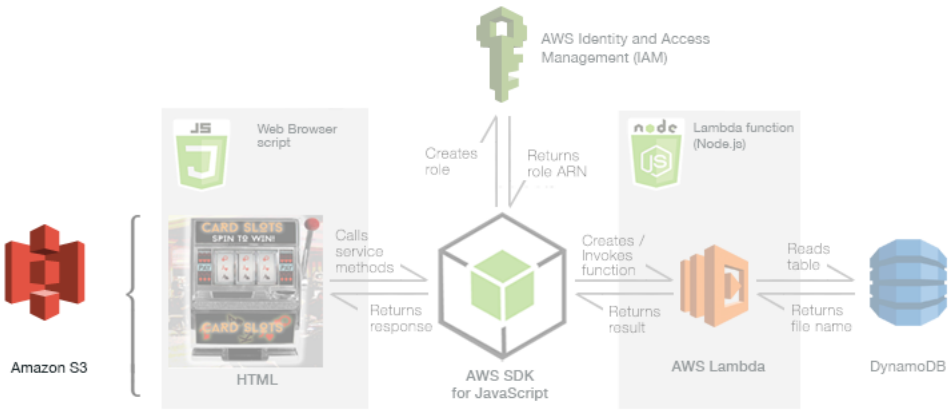
   On Linux or Mac let's use `~/MyLambdaApp`; on Windows let's use `C:\MyLambdaApp`. From now on we'll just call it `MyLambdaApp`.
2. Download `slotassets.zip` from the code example archive on GitHub. This archive contains the browser assets that are used by the application, the Node.js code that's used in the Lambda function, and several setup scripts. In this tutorial, you modify the `index.html` file and upload all the browser asset files to an Amazon S3 bucket you provision for this application. As part of creating the Lambda function, you also modify the Node.js code in `slotpull.js` before uploading it to the Amazon S3 bucket.

   Unzip the contents of `slotassets.zip` as the directory `slotassets` in `MyLambdaApp`. The `slotassets` directory should contain the 30 files.
3. Create a JSON file with your account credentials in your working directory. This file is used by the setup scripts to authenticate their AWS requests. For details, see Loading Credentials in Node.js from a JSON File (p. 32).

4. Create an Amazon S3 bucket configured as a static website (p. 241).
5. Prepare the browser script (p. 242). Save the edited copy of `index.html` for upload to Amazon S3.
6. Create a Lambda execution role in IAM (p. 244).
7. Create and populate an Amazon DynamoDB table (p. 246).
8. Prepare and create the Lambda function (p. 249).
9. Run the Lambda function. (p. 252)

Create an Amazon S3 Bucket Configured as a Static Website

In this task, you create and prepare the Amazon S3 bucket used by the application.



For this application, the first thing you need to create is an Amazon S3 bucket to store all the browser assets. These include the HTML file, all graphics files, and the CSS file. The bucket is configured as a static website so that it also serves the application from the bucket's URL.

The `slotassets` directory contains the Node.js script `s3-bucket-setup.js` that creates the Amazon S3 bucket and sets the website configuration.

To create and configure the Amazon S3 bucket that the tutorial application uses

- At the command line, type the following command, where *`BUCKET_NAME`* is the name for the bucket:

  ```
  node s3-bucket-setup.js BUCKET_NAME
  ```

  The bucket name must be globally unique. If the command succeeds, the script displays the URL of the new bucket. Make a note of this URL because you'll use it later.

Setup Script

The setup script runs the following code. It takes the command-line argument that is passed in and uses it to specify the bucket name and the parameter that makes the bucket publicly readable. It then sets up the parameters used to enable the bucket to act as a static website host.

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

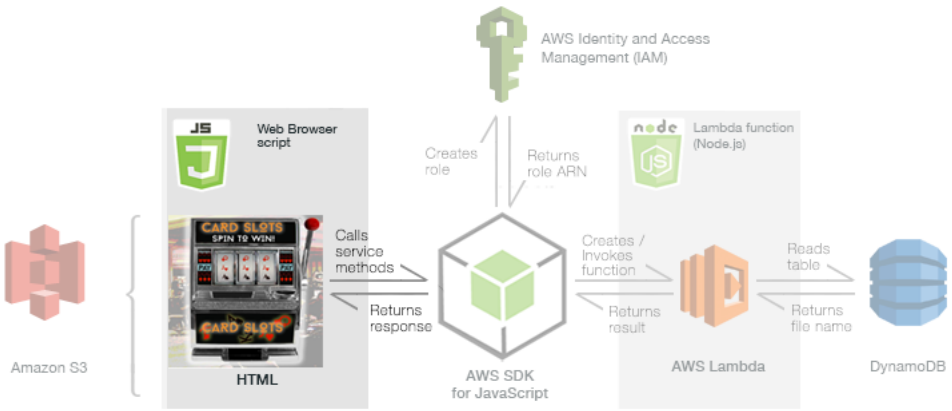
```
var AWS = require('aws-sdk');
// Load credentials and set Region from JSON file
AWS.config.loadFromPath('./config.json');

// Create S3 service object
s3 = new AWS.S3({apiVersion: '2006-03-01'});

// Create params JSON for S3.createBucket
var bucketParams = {
  Bucket : process.argv[2],
  ACL : 'public-read'
};

// Create params JSON for S3.setBucketWebsite
var staticHostParams = {
  Bucket: process.argv[2],
  WebsiteConfiguration: {
  ErrorDocument: {
    Key: 'error.html'
  },
  IndexDocument: {
    Suffix: 'index.html'
  },
  }
};

// Call S3 to create the bucket
s3.createBucket(bucketParams, function(err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Bucket URL is ", data.Location);
    // Set the new policy on the newly created bucket
    s3.putBucketWebsite(staticHostParams, function(err, data) {
      if (err) {
        // Display error message
        console.log("Error", err);
      } else {
        // Update the displayed policy for the selected bucket
        console.log("Success", data);
      }
    });
  }
});
```

Click **next** to continue the tutorial.

Prepare the Browser Script

This topic is part of a larger tutorial about using the AWS SDK for JavaScript with AWS Lambda functions. To start at the beginning of the tutorial, see Tutorial: Creating and Using Lambda Functions (p. 239).

In this task, you will focus on creating an Amazon Cognito identity pool used to authenticate your browser script code, and then editing the browser script accordingly.



Prepare an Amazon Cognito Identity Pool

The JavaScript code in the browser script needs authentication to access AWS services. Within webpages, you typically use Amazon Cognito Identity to do this authentication. First, create an Amazon Cognito identity pool.

To create and prepare an Amazon Cognito identity pool for the browser script

1. Open the Amazon Cognito console, choose **Manage Federated Identities**, and then choose **Create new identity pool**.
2. Enter a name for your identity pool, choose **enable access to unauthenticated identities**, and then choose **Create Pool**.
3. Choose **View Details** to display details on both the authenticated and unauthenticated IAM roles created for this identity pool.
4. In the summary for the unauthenticated role, choose **View Policy Document** to display the current role policy.
5. Choose **Edit** to change the role policy, and then choose **Ok**.
6. In the text box, edit the policy to insert this `"lambda:InvokeFunction"` action, so the full policy becomes the following.

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

7. Choose **Allow**.
8. Choose **Sample code** in the side menu. Make a note of the identity pool ID, shown in red text in the console.

Edit the Browser Script

Next, update the browser script to include the Amazon Cognito identity pool ID created for this application.

To prepare the browser script in the webpage

1. Open `index.html` in the `MyLambdaApp` folder in a text editor.
2. Find this line of code in the browser script.

   ```
   AWS.config.credentials = new AWS.CognitoIdentityCredentials({IdentityPoolId:
   'IDENTITY_POOL_ID'});
   ```
3. Replace *`IDENTITY_POOL_ID`* with the identity pool ID you obtained previously.
4. Save `index.html`.

Click **next** to continue the tutorial.

Create a Lambda Execution Role in IAM

This topic is part of a larger tutorial about using the AWS SDK for JavaScript with AWS Lambda functions. To start at the beginning of the tutorial, see Tutorial: Creating and Using Lambda Functions (p. 239).

In this task, you will focus on creating IAM role used by the application to execute the Lambda function.

A Lambda function requires an execution role created in IAM that provides the function with the necessary permissions to run. For more information about the Lambda execution role, see Manage Permissions: Using an IAM Role (Execution Role) in the *AWS Lambda Developer Guide*.

To create the Lambda execution role in IAM

1. Open `lambda-role-setup.js` in the `slotassets` directory in a text editor.
2. Find this line of code.

```
const ROLE = "ROLE"
```

Replace *ROLE* with another name.

3. Save your changes and close the file.
4. At the command line, type the following.

```
node lambda-role-setup.js
```

5. Make a note of the ARN returned by the script. You need this value to create the Lambda function.

Setup Script Code

The following code is the setup script that creates the Lambda execution role. The setup script creates the JSON that defines the trust relationship needed for a Lambda execution role. It also creates the JSON parameters for attaching the AWSLambdaRole managed policy. Then it assigns the string version of the JSON to the parameters for the `createRole` method of the IAM service object.

The `createRole` method automatically URL-encodes the JSON to create the execution role. When the new role is successfully created, the script displays its ARN. Then the script calls the `attachRolePolicy` method of the IAM service object to attach the managed policy. When the policy is successfully attached, the script displays a confirmation message.

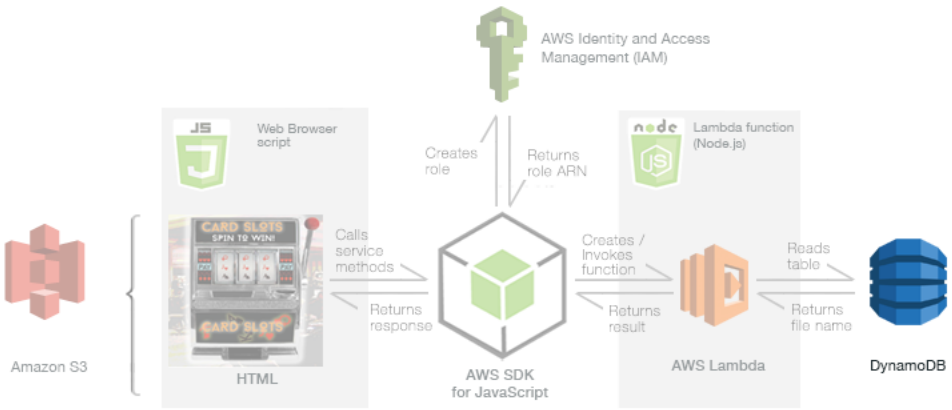
```
// Load the AWS SDK for Node.js
var AWS = require('aws-sdk');
// Load credentials and set Region from JSON file
AWS.config.loadFromPath('./config.json');

// Create the IAM service object
var iam = new AWS.IAM({apiVersion: '2010-05-08'});

var myPolicy = {
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
};

var createParams = {
 AssumeRolePolicyDocument: JSON.stringify(myPolicy),
 RoleName: "ROLE"
};

var policyParams = {
 PolicyArn: "arn:aws:iam::policy/service-role/AWSLambdaRole",
 RoleName: "ROLE"
};

iam.createRole(createParams, function(err, data) {
  if (err) {
     console.log(err, err.stack); // an error occurred
  } else {
     console.log("Role ARN is", data.Role.Arn);           // successful response
     iam.attachRolePolicy(policyParams , function(err, data) {
     if (err) {
        console.log(err, err.stack);
```

```
    } else {
       console.log("AWSLambdaRole Policy attached");
    }
  });
 }
});
```

Click **next** to continue the tutorial.

Create and Populate a DynamoDB Table

In this task, you create and populate the DynamoDB table used by the application.



The Lambda function generates three random numbers, then uses those numbers as keys to look up file names stored in an Amazon DynamoDB table. In the `slotassets.zip` archive file are two Node.js scripts named `ddb-table-create.js` and `ddb-table-populate.js`. Together these files create the DynamoDB table and populate it with the names of the image files in the Amazon S3 bucket. The Lambda function exclusively provides access to the table. Completing this portion of the application requires you to do these things:

- Edit the Node.js code used to create the DynamoDB table.
- Run the setup script that creates the DynamoDB table.
- Run the setup script, which populates the DynamoDB table with data the application expects and needs.

To edit the Node.js script that creates the DynamoDB table for the tutorial application

1. Open `ddb-table-create.js` in the `slotassets` directory in a text editor.
2. Find this line in the script.

   ```
   TableName: "TABLE_NAME"
   ```

   Change *`TABLE_NAME`* to one you choose. Make a note of the table name.
3. Save and close the file.

To run the Node.js setup script that creates the DynamoDB table

- At the command line, type the following.

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

Table Creation Script

The setup script `ddb-table-create.js` runs the following code. It creates the parameters that the JSON needs to create the table. This includes setting the table name, defining the sort key for the table (`slotPosition`), and defining the name of the attribute that contains the file name of one of the 16 PNG images used to display a slot wheel result. It then calls the `createTable` method to create the table.

```
// Load the AWS SDK for Node.js
var AWS = require('aws-sdk');
// Load credentials and set Region from JSON file
AWS.config.loadFromPath('./config.json');

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

var tableParams = {
  AttributeDefinitions: [
    {
      AttributeName: 'slotPosition',
      AttributeType: 'N'
    },
    {
      AttributeName: 'imageFile',
      AttributeType: 'S'
    }
  ],
  KeySchema: [
    {
      AttributeName: 'slotPosition',
      KeyType: 'HASH'
    },
    {
      AttributeName: 'imageFile',
      KeyType: 'RANGE'
    }
  ],
  ProvisionedThroughput: {
    ReadCapacityUnits: 1,
    WriteCapacityUnits: 1
  },
  TableName: 'TABLE_NAME',
  StreamSpecification: {
    StreamEnabled: false
  }
};

ddb.createTable(tableParams, function(err, data) {
  if (err) {
    console.log("Error", err);
  } else {
    console.log("Success", data);

  }
});
```

Once the DynamoDB table exists, you can populate it with the items and data the application needs. The `slotassets` directory contains t Node.js script `ddb-table-populate.js` that automates data population for the DynamoDB table you just created.

To run the Node.js setup script that populates the DynamoDB table with data

1. Open `ddb-table-populate.js` in a text editor.

2. Find this line in the script.

   ```
   var myTable = 'TABLE_NAME';
   ```

   Change *TABLE_NAME* to the name of the table you created previously.
3. Save and close the file.
4. At the command line, type the following.

   ```
   node ddb-table-populate.js
   ```

Table Population Script

The setup script `ddb-table-populate.js` runs the following code. It creates the parameters that the JSON needs to create each data item for the table. These include a unique numeric ID value for `slotPosition` and the file name of one of the 16 PNG images of a slot wheel result for `imageFile`. After setting the needed parameters for each possible result, the code repeatedly calls a function that executes the `putItem` method to populate items in the table.

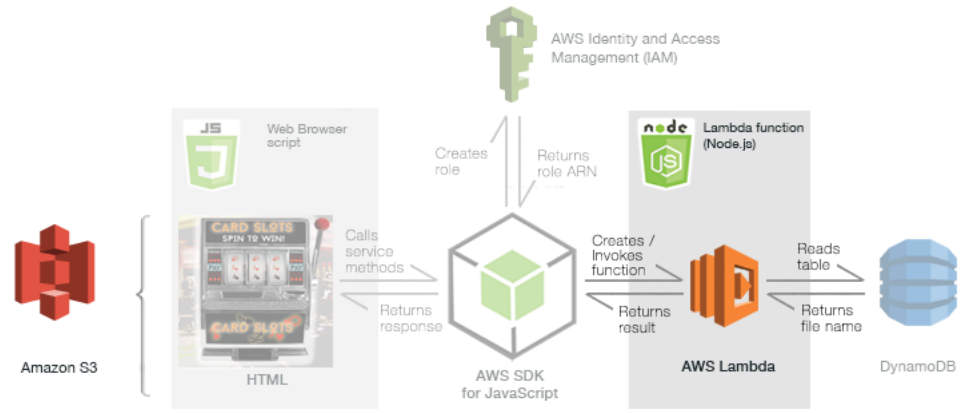
```
// Load the AWS SDK for Node.js
var AWS = require('aws-sdk');
// Load credentials and set Region from JSON file
AWS.config.loadFromPath('./config.json');

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

var myTable = 'TABLE_NAME';

// Add the four results for spades
var params = {
  TableName: myTable,
  Item: {'slotPosition' : {N: '0'}, 'imageFile' : {S: 'spad_a.png'}
  }
};
post();

var params = {
  TableName: myTable,
  Item: {'slotPosition' : {N: '1'}, 'imageFile' : {S: 'spad_k.png'}
  }
};
post();

var params = {
  TableName: myTable,
  Item: {'slotPosition' : {N: '2'}, 'imageFile' : {S: 'spad_q.png'}
  }
};
post();

var params = {
  TableName: myTable,
  Item: {'slotPosition' : {N: '3'}, 'imageFile' : {S: 'spad_j.png'}
  }
};
post();

// Add the four results for hearts
.
.
.
```

```
// Add the four results for diamonds
.
.
.

// Add the four results for clubs
var params = {
  TableName: myTable,
  Item: {'slotPosition' : {N: '12'}, 'imageFile' : {S: 'club_a.png'}
  }
};
post();

var params = {
  TableName: myTable,
  Item: {'slotPosition' : {N: '13'}, 'imageFile' : {S: 'club_k.png'}
  }
};
post();

var params = {
  TableName: myTable,
  Item: {'slotPosition' : {N: '14'}, 'imageFile' : {S: 'club_q.png'}
  }
};
post();

var params = {
  TableName: myTable,
  Item: {'slotPosition' : {N: '15'}, 'imageFile' : {S: 'club_j.png'}
  }
};
post();


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

Click **next** to continue the tutorial.

Prepare and Create the Lambda Function

In this task, you create the Lambda function used by the application.



The Lambda function is invoked by the browser script every time the player of the game clicks and releases the handle on the side of the machine. There are 16 possible results that can appear in each slot position, chosen at random.

The Lambda function generates three random results, one for each slot wheel in the game. For each result, the Lambda function calls the Amazon DynamoDB table to retrieve the file name of the result graphic. Once all three results are determined and their matching graphic URLs are retrieved, the result information is returned to the browser script for showing the result.

The Node.js code needed for the Lambda function is in the `slotassets` directory. You must edit this code to use it in your Lambda function. Completing this portion of the application requires you to do these things:

- Edit the Node.js code used by the Lambda function.
- Compress the Node.js code into a .zip archive file that you then upload to the Amazon S3 bucket used by the application.
- Edit the Node.js setup script that creates the Lambda function.
- Run the setup script, which creates the Lambda function from the .zip archive file in the Amazon S3 bucket.

To make the necessary edits in the Node.js code of the Lambda function

1. Open `slotpull.js` in the `slotassets` directory in a text editor.
2. Find this line of code in the browser script.

```
TableName: = "TABLE_NAME";
```

3. Replace *`TABLE_NAME`* with your DynamoDB table name.
4. Save and close the file.

To prepare the Node.js code for creating the Lambda function

1. Compress `slotpull.js` into a .zip archive file for creating the Lambda function.
2. Upload `slotpull.js.zip` to the Amazon S3 bucket you created for this app. You can use the following CLI command, where *`BUCKET`* is the name of your Amazon S3 bucket:

```
aws s3 cp slotpull.js.zip s3://BUCKET
```

3. Upload the website files - the HTML, PNG, and CSS files - to the bucket.

Lambda Function Code

The code creates a JSON object to package the result for the browser script in the application. Next, it generates three random integer values that are used to look up items in the DynamoDB table, and obtains file names of images in the Amazon S3 bucket. The result JSON is populated and passed back to the Lambda function caller.

Creating the Lambda Function

You can provide the Node.js code for the Lambda function is in a file compressed into a .zip archive file that you upload to an Amazon S3 bucket. The `slotassets` directory contains the Node.js script `lambda-function-setup.js` that you can modify and run to create the Lambda function.

To edit the Node.js setup script for creating the Lambda function

1. Open `lambda-function-setup.js` in the `slotassets` directory in a text editor.
2. Find this line in the script

   ```
   S3Bucket: 'BUCKET_NAME',
   ```

   Replace *BUCKET_NAME* with the name of the Amazon S3 bucket that contains the .zip archive file of the Lambda function code.
3. Find this line in the script

   ```
   S3Key: 'ZIP_FILE_NAME',
   ```

   Replace *ZIP_FILE_NAME* with the name of the .zip archive file of the Lambda function code in the Amazon S3 bucket.
4. Find this line in the script.

   ```
   Role: 'ROLE_ARN',
   ```

   Replace *ROLE_ARN* with the ARN of the execution role you just created.
5. Save and close the file.

To run the setup script and create the Lambda function from the .zip archive file in the Amazon S3 bucket

- At the command line, type the following.

  ```
  node lambda-function-setup.js
  ```

Creation Script Code

The following is the script that creates the Lambda function. The code assumes you uploaded the .zip archive file of the Lambda function to the Amazon S3 bucket you created for the application.

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

```
var AWS = require('aws-sdk');
// Load credentials and set Region from JSON file
AWS.config.loadFromPath('./config.json');

// Create the IAM service object
var lambda = new AWS.Lambda({apiVersion: '2015-03-31'});

var params = {
  Code: { /* required */
    S3Bucket: 'BUCKET_NAME',
    S3Key: 'ZIP_FILE_NAME'
  },
  FunctionName: 'slotTurn', /* required */
  Handler: 'slotSpin.Slothandler', /* required */
  Role: 'ROLE_ARN', /* required */
  Runtime: 'nodejs8.10', /* required */
  Description: 'Slot machine game results generator'
};
lambda.createFunction(params, function(err, data) {
  if (err) console.log(err); // an error occurred
  else     console.log("success");           // successful response
});
```

Click **next** to continue the tutorial.

Run the Lambda Function

In this task, you run the application.

To run the browser application

1. Open a web browser.
2. Open the `index.html` in the Amazon S3 bucket that hosts the application.

   Note
   To do this, go open the Amazon S3 bucket in the AWS console, select the bucket, and select the **Object URL**.

3. Select the handle on the right side of the slot machine. The wheels begin to spin as the browser script invokes the Lambda function to generate results for this turn.

4. Once the Lambda function returns the spin results to the browser script, the browser script sets the game display to show the images that the Lambda function selected.
5. Select the handle again to start another spin.

Tutorial Clean Up

To avoid ongoing charges for the resources and services used in this tutorial, delete the following resources in their respective service consoles:

- The Lambda function in the AWS Lambda console at https://console.aws.amazon.com/lambda/.
- The DynamoDB table in the Amazon DynamoDB console at https://console.aws.amazon.com/dynamodb/.
- The objects in the Amazon S3 bucket and the bucket itself in the Amazon S3 console at https://console.aws.amazon.com/s3/.
- The Amazon Cognito identity pool in the Amazon Cognito console at https://console.aws.amazon.com/cognito/.
- The Lambda execution role in the IAM console at https://console.aws.amazon.com/iam/.

Congratulations! You have now finished the tutorial.