AWS

Lambda Guide Book

By Gordon Wong

AWS Lambda Guide Book

By Gordon Wong

Table of Contents

Disclaimer

While all attempts have been made to verify the information provided in this book, the author does assume any responsibility for errors, omissions, or contrary interpretations of the subject matter contained within. The information provided in this book is for educational and entertainment purposes only. The reader is responsible for his or her own actions and the author does not accept any responsibilities for any liabilities or damages, real or perceived, resulting from the use of this information.

The trademarks that are used are without any consent, and the publication of the trademark is without permission or backing by the trademark owner. All trademarks and brands within this book are for clarifying purposes only and are the owned by the owners themselves, not affiliated with this document.

Introduction

The Amazon AWS Lambda has become popular of late, and it is good for you to know how to work with it. When using it, there are limits imposed on the use of the various resources, and there is a need for you to know how to work with these. Permissions are also necessary for accessing the various resources, as well as for the purpose of cross-account access. For instance, for you to use the AWS Lambda console, your account must have been granted some permissions. This guide explores all these. Enjoy reading!

Chapter 1- Creation of a Deployment Package

For us to create a Lambda function, we have to first create a deployment package, which should be a “*.jar*” or “*.zip*” file having the code and the dependencies you have created. The zip should have only the code and its dependencies, but not the containing folder. We need to demonstrate how this can be done in Node.js, Python, and Java.

Node.js

You can choose to write a deployment package on your own or write the code in the Lambda console directly, in which the console will take over the role of creating the deployment package for you and then upload it, meaning that you will have created your Lambda function.

The steps given below can be followed for you to create a deployment package, but outside the console. In our case, you are need to create a deployment package which features the file named "*myfile.js"* and then makes use of the "*async"* library.

1. Launch your text editor, and then write your code in it. Save the file with the name "*myfile.js."* The file name will be used for specifying the handler when you are creating the lambda function.
2. Make use of the "*npm"* package so as to install the libraries which your code will need. An example is when your code is to use the "*async"* library. In such a case, the following "*npm"* command can be used:

   npm install async

3. The following should be the current structure of your library:

filename.js

node_modules/async

node_modules/async/lib

node_modules/async/lib/async.js

node_modules/async/package.json

4. The contents of the folder should then be zipped, and this should form your deployment package.

After that, it is good for you to specify the name of the .zip file as the deployment package, and this should be during the time of creation of the deployment of the package.

If you are in need of including your own libraries, it will be good for you to include the native ones. You can do that by packing them in the zip file and then referencing them during the process of calling them from the Node.js or from the processes which you have started previously. The following lines should be included at the beginning of your

function code:

process.env[‘PATH’] = process.env[‘PATH’] + ‘:’ +
process.env[‘LAMBDA_TASK_ROOT’]

Java

In this case, the deployment package can be in the form of a standalone jar file or a .zip file. One can use maven to package their Java code at the command line to get a deployment package. Before going further, it will be good for us to start by installing the built tool for the maven command line. In Linux, execute the following command:

sudo apt-get install mvn

For those using Homebrew, execute the following command:

brew install maven

The following should be the structure of the folder once you have setup the project:

project-dir/pom.xml

project-dir/src/main/java/ (add your code here)

Your code will then be located in the /java folder. An example is when your package is named "*sample,*" and your class is named "*Hello.js.*" The structure will be as shown below:

project-dir/src/main/java/sample/Hello.java

Once the project has been built, the .jar file will be located in the subdirectory "*project-dir*/target."

Follow the steps given below:

1. Create the project
    - The first step in this should be creating a directory for the project, which is "*project-dir.*"

- Create the following in the above directory:

The file for the project object mode, "*pom.xml.*" The following information for the project and configuration details should be added for the Maven to build our project:

<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>doc-samples</groupId>

<artifactId>lambda-java-sample</artifactId>

<packaging>jar</packaging>

<version>1.0-SNAPSHOT</version>

<name>lambda-java-sample</name>

<dependencies>

<dependency>

```
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-lambda-java-core</artifactId>
    <version>1.1.0</version>
    </dependency>
</dependencies>

<build>
    <plugins>
    <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-shade-plugin</artifactId>
    <version>2.3</version>
    <configuration>
    <createDependencyReducedPom>false</createDependencyReducedPom>
    </configuration>
    <executions>
    <execution>
    <phase>package</phase>
    <goals>
    <goal>shade</goal>
    </goals>
```

```
    </execution>
    </executions>
    </plugin>
    </plugins>
  </build>
</project>
```

In the section for “*dependencies,*” the property “*groupID*” is the ID for Amazon AWS in the Maven Central Repository.

- The following structure should then be created in the “*project-dir*”:

  ***project-dir*/src/main/java**

  Under the subdirectory *“/java,”* add the java files and the folder structure, if you have any. If the name for the java package is “*sample,*” and the source code is “*Hello.java,*” the directory structure will look as follows:

***project-dir*/src/main/java/sample/Hello.java**

2. Building the project.

This is where the deployment package will be created. We will use the Maven so as to build our project at the command line. The following steps should be followed:

- Use the necessary commands to change your directory on the command line to the directory of the project (*project-dir).*

- Use the following Maven command so as to build your project. Here is the command:

$ mvn package

A .jar file will be created from the above command, and this will be saved in the directory "*project-dir*/target/lambda-java-sample-1.0-SNAPSHOT.jar."

The build will then have created a .jar file from the information provided by the file "*pom.xml"* so as to perform any necessary transforms. The file will have all of the necessary dependencies. You will then have your deployment package which you can add to the AWS Lambda for creation of a Lambda function.

Note that in our above case, we have not used any IDE so as to create a deployment package. We want to use an IDE, and demonstrate how this can be done.

Use of Maven and Eclipse IDE (Java) for creation of a .jar Deployment Package

One can use their Eclipse IDE for java so as to package their code into a deployment package.

Before beginning to do this, we should install a Maven plugin for the eclipse. The following steps are necessary:

1. Launch eclipse. Click on the "*Help*" menu in eclipse, and then choose the option for "*Install New Software.*"

2. In the window for "*Install,*" add the text

3. **http://download.eclipse.org/technology/m2e/releases** and Work with: box, then click on "*Add.*"

4. Make sure that you have completed the above steps.

After that, we can begin to work with our project so as to create the deployment package. The following steps can be followed:

1. Creating and building the project.

We should launch the eclipse for creation of a Maven project. The necessary dependencies should also be added and then the project built. The .jar file which results from this will be our deployment package.

1. We need to create a Maven project in the eclipse.

2. Choose “New” from the “File” menu, and then select “Project.”

3. In the window for “New Project,” select “Maven project.”

4. In the window for “New Maven Project”, select “Create a simple project” and then leave the default selections as they are.

5. In the windows for “New Maven Project,” “Configure project,” type the artifact information given below:

- Group Id: doc-samples

- Artifact Id: lambda-java-sample

- Version: 0.0.1-SNAPSHOT

- Packaging: jar

- Name: lambda-java-sample

2. Add the dependency “aws-lambda-java-core” into the file “pom.xml.” This will provide the project with definitions for the interfaces RequestHandler, RequestStreamHandler, and Context. With that, you will be in a position to compile the code so that it can be used in AWS Lambda.

1. Launch the context menu for the file pom.xml, select “Maven,” and then “Add Dependency.”

2. The following should be typed in the window for “Add dependency”:

- Group Id: com.amazonaws
- Artifact Id: aws-lambda-java-core
- Version: 1.1.0

3. Add a new java class to your project.

    1. Launch the context menu for the subdirectory “the src/main/java” in your project, then select “New” followed by “Class.”

    2. In the window for “New Java Class,” type the values given below:

- Package: example

- Name: Hello

3. Add the java code to the file.

4. Build the project.

In the Package Explorer, launch the context menu for your project. Click on "*Run AS"* and then choose "*Maven Build…."* In the Window for "*Edit Configuration,"* identify the "*Goals"* box, and then type the package in it.

5. Add the plugin "maven-shade-plugin," and then rebuild the project.

The plugin will take the artifacts which are produced by the package goal, and a

standalone .jar file will be created having the compiled customer code, and the dependencies which have been resolved from the file pom.xml.

1. Launch the context menu for the file pom.xml, select “Maven,” and then “Add plugin.”

2. In the window for “Add Plugin,” type the values given below:

- Group Id: org.apache.maven.plugins

- Artifact Id: maven-shade-plugin
- Version: 2.3

3. You can then build the project again. During this time, the jar will be created as before, and then make use of the plugin “aven-shade-plugin” so as to pull the dependencies for the purpose of making a standalone .jar file.

Launch the context menu for your project, select “Run As” followed by “Maven build…”

In the windows for “Edit Configurations,” identify the “Goals” box and then in it, type the text “shade:shade.”

- Select “Run.”

The resulting .jar file can be found in the subdirectory “/target.” You can then open the context menu for this subdirectory, select “Show In,” and then select “System Explorer.” This is where you will find the file “lambda-java-example-0.0.1-SNAPSHOT.jar,” which is your deployment package.

Creation of a .zip Deployment Package

You have now learned how to create a .jar deployment package for your project. In this section, we will explore how to create a .zip deployment package from your java project. This can be done by use of any building and packaging tool. However, regardless of the tool which you use to do this, the .zip file that results must have the structure given below:

- All the compiled resource files and class files should be located at the root level.

- All the jar files required for running the code should be located at the /lib directory.

Before we can be able to do anything, you should first download Gradle.

Using Gradle and Maven Central repository for creation of .zip file

We need to end with a project directory (*project-dir)* with the structure given below:

- project-dir/build.gradle

- project-dir/src/main/java/

Your code will be located in the /java folder. If the name of your package is "*sample,"* and the java class "*Hello.java"* is located in it, then you will have the following:

***project-dir*/src/main/java/sample/Hello.java**

Once the project has been built, you will find the final .zip file in the subdirectory

"*project-dir*/build/distributions."

1. Begin by creating the project directory (project-dir).

2. In the above directory, create the file *"build.gradle"* and then add the following code into the file:

```
apply plugin: 'java'
repositories {
    mavenCentral()
}

dependencies {
    compile (
    'com.amazonaws:aws-lambda-java-core:1.1.0',
    'com.amazonaws:aws-lambda-java-events:1.1.0'
    )
}
```

```
task buildZip(type: Zip) {
    from compileJava
    from processResources
    into('lib') {
    from configurations.runtime
    }
}

build.dependsOn buildZip
```

3. Create the structure given below in the directory "*project-dir)":

***project-dir*/src/main/java/**

4. Under the subdirectory /java, add your java files and the folder structure, if you have any. If you have a java package by the name "*sample*" and a source code named "*Hello.java,*" then the structure of the directory should appear as shown below:

***project-dir*/src/main/java/sample/Hello.java**

5. Run the Gradle command given below so as to package and build the project and get a .zip file:

***project-dir*> gradle build**

6. You can then verify the "*project-dir.zip"* file which results in the subdirectory "*project-dir*/build/distributions."

7. At this point, the .zip file, which is the deployment package, can be uploaded to the AWS Lambda for creation of a Lambda function and then you can invoke it by use of a sample event data for the purpose of testing it.

Using Local jars and Gradle for creation of a .zip file

It is not a must for you to use the Maven Central Repository. You can choose to add all your dependencies into the project folder. In such a case, the project folder should have the directory structure which is shown below:

***project-dir*/jars/ (all jars should be added here)**

***project-dir*/build.gradle**

***project-dir*/src/main/java/ (your code should be added here)**

So, if our Java code is having the package "*sample,*" and the class "*Hello.java,*" the code will be located in the subdirectory given below:

***project-dir*/src/main/java/sample/Hello.java**

The file "*build.gradle*" should have the contents given below:

```
apply plugin: ‘java’

dependencies {
    compile fileTree(dir: ‘jars’, include: ‘*.jar’)
}

task buildZip(type: Zip) {
    from compileJava
    from processResources
    into(‘lib’) {
    from configurations.runtime
    }
}

build.dependsOn buildZip
```

Remember that the dependencies we have are specifying the “*fileTree”* which will identify the “*project-dir*/jars” as our subdirectory which will have all the jars which are required.

It is now time for the package to be built. The following Gradle command can be executed for this purpose:

***project-dir*> gradle build**

That is it.

Python

You have now learned how to create a deployment in Node.js and as well as in Java. Now, we need to explore how this can be done in Python.

For the Lambda function to be created, we should first create a deployment package for the Lambda function, which should be a .zip file having all the code and dependencies.

You can choose to create your deployment package yourself, or write the code directly in the console, whereby the console will create the deployment package on your behalf and then upload it, and a Lambda function will be created.

We will have to use "*pip*" for the purpose of installing the dependencies/libraries. After the installation of "*pip,*" follow the steps given below:

1. Create a directory for the project, such as "*project-dir.*"

2. Save all the source files for Python at the root level of the directory you have created. The source files are the ones with a .py extension.

3. Use pip to install any required libraries. Also, these libraries should be installed at the root level of the directory you have just created. The following command can be used for that purpose:

pip install *module-name* -t */path/to*/project-dir

Consider the command given below, which can be used for installation of the library "*requests*" HTTP in the directory "*project-dir.*"

pip install requests -t */path/to*/project-dir

4. The contents of the directory "*project-dir*" can then be zipped, and this will give you the deployment package. However, it is good for you to know that you should zip the

contents of the directory, but not the directory itself. The contents contained in the zip file will be available as our current working directory in the Lambda function.

Sometimes, you might have used the virtualenv tool for creation of a Python environment.

The following steps are necessary for creation of a deployment package:

1. Use your Python code to create a .zip file which you need to upload to the AWS Lambda.

2. The libraries from the preceding activated virtual environment should then be added. This means that you should add the contents of the directory to the .zip file. If you are using the Windows OS, the directory should be the following:

%VIRTUAL_ENV%\Lib\site-packages

For OS X and Linux users, the directory should be the following:

$VIRTUAL_ENV/lib/python2.7/site-packages

That is what we can say as far as this is concerned. In some cases, you will fail to get the directory. If this happens with you, just find it in the directory "*dist-packages.*".

Suppose that you had readily installed the package "*requests*" in the environment which has been activated, assuming that these will be used in your code. The packages can be installed as follows:

pip install requests

The following steps can then be followed for the purpose of creation of the deployment package:

1. From the Python code which is to be uploaded to the AWS Lambda, create a .zip file.
2. Add the libraries from which you had previously activated a virtual environment to

your .zip file. This means that the contents of the following directory should be added to the .zip file:

In Windows, here is the directory:

%VIRTUAL_ENV%\Lib\site-packages

For Linux and OS X, the directory is:

$VIRTUAL_ENV/lib/python2.7/site-packages

Chapter 2- AWS Lambda Limits

Every function in Lambda is allocated a specified amount of resources regardless of the amount of memory allocated, and each function has to be allocated a fixed amount of code storage for each function and per count.

Sometimes, you might have a concurrent execution, and in such a case, you can request an increase for the limit. This can be done by following the steps given below:

- Begin by opening the page for AWS Support Center, log in, and then click on "*Create Case.*"

- Choose "*Service Limit Increase*" under "*Regarding.*"

- Select "*Lambda*" from "*Limit Type,*" and then fill in any necessary fields provided in the form, and then select the best method for contact at the bottom of your page.

Sometimes, the AWS might automatically raise the limit for your concurrent execution so that the function can be enabled to meet the event rate which is incoming, such as when you are triggering a function from the S3 bucket for Amazon.

AWS Lambda Limit Errors

In case the function that you define exceeds the specified limit for a particular resource, then it usually fails with an exception of type “exceeded limits.” The limit is fixed, and changing it at this time will be impossible. Consider a situation in which you receive an exception of type “CodeStorageExceededException,” or an error of the type “Code storage limit exceeded.” For you to solve that problem, you will have to make a reduction on your code storage. This can be done by following the steps given below:

Remove the functions which you don’t need to use anymore.

Reduce the size of the code for the functions which you do not need to remove. The size of code for a Lambda function can be found by use of the AWS Lambda console, which is the command line interface for AWS, or the AWS SDKs.

Chapter 3- Event Sources in AWS Lambda

Invocation of AWS Lambda functions is done by the other AWS services which are responsible for publishing events. A Lambda function is invoked after the configuration of the supported service as an event source.

We want to discuss the AWS services supported by Lambda.

CloudFormation Create Request

{

“StackId”: ***myid,***

“ResponseURL”: “http://pre-signed-S3-url-for-response”,

“ResourceProperties”: {

“StackName”: “name”,

“List”: [

“1”,

“2”,

“3”

]

},

“RequestType”: “Create”,

“ResourceType”: “Custom::TestResource”,

“RequestId”: “This is a unique for the create request”,

“LogicalResourceId”: “TestResource”

}

SES Email Receiving

“Records”: [

{

“eventVersion”: “1.0”,

“ses”: {

“mail”: {

“commonHeaders”: {

“from”: [

“John Joel <johnjoel@sample.com>”

],

“to”: [

“johnjoel@sample.com”

],

“returnPath”: “johnjoel@sample.com”,

“messageId”: “<0724365475sample.com>”,

“date”: “Fri, 6 May 2016 11:23:56 -0800”,

“subject”: “Subject Exam”

},

“source”: “johnjoel@sample.com”,

“timestamp”: “1990-02-03T00:00:00.000Z”,

“destination”: [

“johnjoel@sample.com”

],

“headers”: [

{

“name”: “Return-Path”,

“value”: “<johnjoel@sample.com>”

},

{

“name”: “Received”,

“value”: “(UTC)”

},

{

“name”: “DKIM-Signature”,

“value”: “v=1; a=rsa-sha256; c=relaxed/relaxed; d=sample.com; s=sample; h=mime-version:from:date:message-id:subject:to:content-type; bh=jX2GohCAI7sIbkHxx3mLYO19ieDQz1R0P7HwQjjlFj3x=; “

},

{

“name”: “MIME-Version”,

“value”: “1.0”

},

{

“name”: “From”,

“value”: “John Joel <johnjoel@sample.com>”

},

{

“name”: “Date”,

“value”: “Wed, 7 Oct 2015 12:34:56 -0700”

},

{

“name”: “Message-ID”,

“value”: “<0724356745sample.com>”

},

{

“name”: “Subject”,

“value”: “Test Subject”

},

{

“name”: “To”,

value”: “johnjoel@sample.com”

},

{

“name”: “Content-Type”,

“value”: “text/plain; charset=UTF-8”

}

],

“headersTruncated”: false,

“messageId”: “12vrnil0j7ic28thm7dfhrd6v0clambdw3nbp0i7x”

},

“receipt”: {

“recipients”: [

“johnjoel@sample.com”

],

“timestamp”: “1990-02-02T00:00:00.000Z”,

“spamVerdict”: {

“status”: “PASS”

},

“dkimVerdict”: {

“status”: “PASS”

},

“processingTimeMillis”: 574,

“action”: {

“type”: “Lambda”,

“invocationType”: “Event”,

“functionArn”: *arn*

},

“spfVerdict”: {

“status”: “PASS”

},

“virusVerdict”: {

“status”: “PASS”

}

}

},

“eventSource”: “aws:ses”

}

]

}]}

Scheduled event

{

“account”: “0732453474754”,

“region”: “us-east-1”,

“detail”: {},

“detail-type”: “Scheduled Event”,

“source”: “aws.events”,

“time”: “1990-02-02T00:00:00Z”,

“id”: “gdc73f7d-aeb9-22d3-9d6a-925b769b0d9d”,

“resources”: [

“arn:aws:events:us-east-1: 0732453474754:rule/my-schedule”

]

}

CloudWatch Logs

{

“awslogs”: {

“data”:

“your data goes here”

}

}

SNS

{

“Records”: [

{

“EventVersion”: “1.0”,

“EventSubscriptionArn”: *eventsub*,

“EventSource”: “aws:sns”,

“Sns”: {

“SignatureVersion”: “1”,

“Timestamp”: “1990-02-02T00:00:00.000Z”,

“Signature”: “SAMPLE”,

“SigningCertUrl”: “SAMPLE”,

“MessageId”: “65df01b5-ee74-5cb3-9987-4c331d41eb4e”,

“Message”: “Hei from SNS!”,

“MessageAttributes”: {

“Test”: {

“Type”: “String”,

“Value”: “TestString”

},

“TestBinary”: {

“Type”: “Binary”,

“Value”: “TestBinary”

}

},

“Type”: “Notification”,

“UnsubscribeUrl”: “SAMPLE”,

“TopicArn”: *arn*,

“Subject”: “TestInvoke”

}

}

]

}

DynamoDB Update

{

“Records”: [

{

“eventID”: “1”,

“eventVersion”: “1.0”,

“dynamodb”: {

“Keys”: {

“Id”: {

“N”: “101”

}

},

“NewImage”: {

“Message”: {

“S”: “Add item!”

},

“Id”: {

“N”: “101”

}

},

“StreamViewType”: “NEW_AND_OLD_IMAGES”,

“SequenceNumber”: “234”,

“SizeBytes”: 26

},

“awsRegion”: “asia-west-2”,

“eventName”: “INSERT”,

“eventSourceARN”: *arn*,

“eventSource”: “aws:dynamodb”

},

{

“eventID”: “2”,

“eventVersion”: “1.0”,

“dynamodb”: {

“OldImage”: {

“Message”: {

“S”: “Add item!”

},

“Id”: {

“N”: “234”

}

},

“SequenceNumber”: “444”,

“Keys”: {

“Id”: {

“N”: “234”

}

},

“SizeBytes”: 59,

“NewImage”: {

“Message”: {

“S”: “The item has been changed”

},

“Id”: {

“N”: “234”

}

},

“StreamViewType”: “NEW_AND_OLD_IMAGES”

},

“awsRegion”: “asia-west-2”,

“eventName”: “MODIFY”,

“eventSourceARN”: *arn*,

“eventSource”: “aws:dynamodb”

},

{

“eventID”: “3”,

“eventVersion”: “1.0”,

“dynamodb”: {

“Keys”: {

“Id”: {

“N”: “234”

}

},

“SizeBytes”: 38,

“SequenceNumber”: “555”,

“OldImage”: {

“Message”: {

“S”: “The item has been changed”

},

“Id”: {

“N”: “234”

}

},

“StreamViewType”: “NEW_AND_OLD_IMAGES”

},

“awsRegion”: “asia-west-2”,

“eventName”: “REMOVE”,

“eventSourceARN”: *arn,*

“eventSource”: “aws:dynamodb”

}

]

}

Cognito Sync Trigger

{

“datasetName”: “NameOfDataSet”,

“eventType”: “SyncTrigger”,

“region”: “asia-east-1”,

“identityId”: “identityId”,

“datasetRecords”: {

“SampleKey2”: {

“newValue”: “value2”,

“oldValue”: “previousValue2”,

“op”: “replace”

},

“SampleKey1”: {

“newValue”: “Value1”,

“oldValue”: “previousValue1”,

“op”: “replace”

}

},

“identityPoolId”: “identityPoolId”,

“version”: 2

}

Kinesis

“Records”: [

{

“eventID”: “shardId-000000000000:78456345474909850182808974697314458218008467564420254 87”,

“eventVersion”: “1.0”,

“kinesis”: {

“partitionKey”: “partitionKey-3”,

“data”: “SGVjkG8sIHRoaXMhbXMgYSB0ZXK0IDExMj4=”,

“kinesisSchemaVersion”: “1.0”,

“sequenceNumber”: “78456345474909850182808974697314458218008467564420254 87”

},

“invokeIdentityArn”: *arn*,

“eventName”: “aws:kinesis:record”,

“eventSourceARN”: *arn*,

“eventSource”: “aws:kinesis”,

“awsRegion”: “asia-east-1”

```
    }
  ]
}
```

S3 Put

“Records”: [

{

“eventVersion”: “2.0”,

“eventTime”: “1990-02-02T00:00:00.000Z”,

“requestParameters”: {

“sourceIPAddress”: “127.0.0.1”

},

“s3”: {

“configurationId”: “ConfigRule”,

“object”: {

“eTag”: “0123456789abwxy0123456789abcjkl”,

“sequencer”: “0A1B2C3D4E5F678901”,

“key”: “Picture.jpg”,

“size”: 1024

},

“bucket”: {

“arn”: *arn*,

“name”: “sourcebucket”,

“ownerIdentity”: {

“principalId”: “SAMPLE”

}

},

“s3SchemaVersion”: “1.0”

},

“responseElements”: {

“x-amz-id-2”:
“SAMPLE256/8456abcdefghijklambdaisgood/mnoutystuvwxyzABCDEFGH”,

“x-amz-request-id”: “SAMPLE123456789”

},

“awsRegion”: “asia-east-1”,

“eventName”: “ObjectCreated:Put”,

“userIdentity”: {

“principalId”: “SAMPLE”

},

“eventSource”: “aws:s3”

}

]

}

S3 Delete

{

“Records”: [

{

“eventVersion”: “2.0”,

“eventTime”: “1990-02-02T00:00:00.000Z”,

“requestParameters”: {

“sourceIPAddress”: “127.0.0.1”

},

“s3”: {

“configurationId”: “ConfigRule”,

“object”: {

“sequencer”: “0J1C2C3D4K5F677501”,

“key”: “Picture.jpg”

},

“bucket”: {

“arn”: *arn*,

“name”: “sourcebucket”,

“ownerIdentity”: {

“principalId”: “SAMPLE”

}

},

“s3SchemaVersion”: “1.0”

},

“responseElements”: {

“x-amz-id-2”: ”
SAMPLE256/8456abcdefghijklambdaisgood/mnoutystuvwxyzABCDEFGH “,

“x-amz-request-id”: “SAMPLE123456789”

},

“awsRegion”: “asia-east-1”,

“eventName”: “ObjectRemoved:Delete”,

“userIdentity”: {

“principalId”: “SAMPLE”

},

“eventSource”: “aws:s3”

}

]

}

Mobile Backend

{

“operation”: “echo”,

“message”: “Hello there!”

}

Chapter 4- Managing Resource Access Permissions

Each resource in AWS is owned by an account, and permission policies governs the process of creating and accessing the resource. The administrator for an account is allowed to attach the permission policies to the IAM identities, and some of the services are also in support of the attachment of permission policies to the resources.

Resources and Operations in AWS Lambda

The primary resources in AWS Lambda include event source mapping and the Lambda function. An event source mapping is created in the AWS Lambda pull model so as to associate a Lambda function with an event source.

Other types of resources supported in AWS Lambda include "*alias"* and "*version."* However, it is good for you to note that the version and alias can be created in the context of a Lambda function which is readily in existence. These are normally known as "*subresources."* What happens is that the resources are given a unique name in the AWS Lambda which is used for identifying them.

Resource Ownership

The resource owner should be the AWS account in which the resource was created. This means that the resource owner should be the AWS account of our principal entity, which can either be a root account, IAM user, or IAM role, and it is tasked with the responsibility of authenticating a request and creating a particular resource. This works as it has been illustrated below:

- If you make use of the root credentials of your root account so as to create the resource, then that AWS account will become the owner of that resource. This translates to the fact that a resource belongs to the account in which it was created.

- After the creation of an IAM user in the AWS account, and then you grant permissions to a user to create a Lambda function, the user will then be granted permission to create the Lambda function. However, the user belongs to your AWS account, meaning that you will own that Lambda function.
- After creation of an IAM role in your AWS account with permissions which can allow you to create a Lambda function, anyone with the ability to assume the role will be in a position to create the Lambda function. The AWS account, which owns the role, will be in possession of the Lambda function as a resource.

A “*permission policy”* is normally used for defining who has access to which resource. Policies which have been attached to an IAM identity are usually known as “*Identity-based policies”* while the policies which have been attached to a particular resource are usually known as “*resource-based policies.”*

Identity-based Policies

Policies can be attached to IAM identities. The following examples best demonstrate this:

- Attach permission policy to a group or particular user in your account. An administrator of a particular account can take advantage of the permissions policy for a particular user to grant permissions for the user so that he or she can be in a position to create a Lambda function.

- Attach a permissions policy to a role (that is, grant cross-account permissions)- with an IAM role, one can choose to attach an identity-based permissions policy to it, which is a good way for granting cross-account permissions.

Suppose that you have two accounts, A and B, and you need to grant a cross-account permission, you can do it as follows:

1. The administrator for account A should create an IAM role, and then attach a

permissions policy to the role which grants permissions on the resources in account A.

2. The administrator for account A then attaches a trust policy to this role, and then makes sure that it identifies the account B as the principal who will assume the role.

3. The administrator for account B will then be in a position to delegate permissions so as to assume the role to any of the users in account B. When this has been done, the users in account B will be in a position to access the resources of account A. You can also choose to use an AWS service principal as the principal in the trust policy if you are in need of granting an AWS service permission for the purpose of assuming the role.

Consider the permission example given below which shows how the permission "*lambda:ListFunctions*" can be granted to the available resources. Currently, the use of "ARNs"for identification of some specific resources is not supported for some of the available API actions, and this is why one is expected to specify the wildcard (*) character.

Here is the policy:

```
{
    “Version”: “2016-05-08”,
    “Statement”: [
    {
    “Sid”: “ListExistingFunctions”,
    “Effect”: “Allow”,
    “Action”: [
    “lambda:ListFunctions”
    ],
    “Resource”: “*”
    }
    ]
}
```

Resource-based Policies

Each function in Lambda can be associated with resource-based permissions policies. A Lambda function forms the primary resource in Lambda, and these types of resources are usually referred to as "*Lambda function policies.*" A Lambda function policy can be used for the purpose of granting cross-account permissions instead of using identity-based policies having IAM roles. An example is that the Amazon S3 can be granted permissions to be in a position to invoke a Lambda function simply by adding permissions to our Lambda function policy instead of having to create an IAM role.

The example given below shows a Lambda policy having a single statement.

```
{
   "Policy":{
   "Version":"2016-05-08",
   "Statement":[
   {
   "Effect":"Allow",
   "Principal":{
```

“Service”:“s3.amazonaws.com”

},

“Action”:“lambda:InvokeFunction”,

“Resource”:“arn:aws:lambda:*region*:*account-id*:function:Hellothere”,

“Sid”:“86bagh70-6a1f-31a8-a8ab-8bb9bc777545”,

“Condition”:{

“StringEquals”:{

“AWS:SourceAccount”:”*account-id*”

},

“ArnLike”:{

“AWS:SourceArn”:“arn:aws:s3:::SampleBucket”

}

}

}

]

}

}

AWS Lambda Access with Identity-Based Policies

Consider the permissions policy given below:

{

“Version”: “2016-08-05”,

“Statement”: [

{

“Sid”: “CreateFunctionPermissions”,

“Effect”: “Allow”,

“Action”: [

“lambda:CreateFunction”

],

“Resource”: “*”

},

{

“Sid”: “PermissionToPassAnyRole”,

“Effect”: “Allow”,

“Action”: [

“iam:PassRole”

],

“Resource”: “arn:aws:iam::*account-id*:role/*”

}

]

}

We have two statements in the policy:

- The first statement has been used for granting permission to the “*lambda:CreateFunction*” which is an AWS action. At this time, AWS Lambda does not support this permission for the action at the resource-level.

- The second statement has been used for granting permission to the IAM action on the IAM role. The * (wildcard character) used at the end of above value for the resource is a symbol for the action “iam:PassRole” on any of the IAM role. If you are in need of limiting the permission to a specified role, the wildcard character should be replaced with a specific name for a role.

The following steps can guide you when creating an IAM role in AWS Lambda:

- Create a user for IAM

  You should begin by creating an IAM user and then add him to a group with the administrative permissions, and then grant this user administrative privileges. After that, you will be in a position to use the credentials of that user and a special url so as to access the AWS account.

- The user can allow the user to list the Lambda Functions

  The IAM user for your account must possess a permission for the action "*lambda:ListFunctions*" so that the user can see anything in the console. After granting the permissions, the console will show the list of Lambda functions in your AWS account which have been created in the specific AWS region in which the user belongs.

{

"Version": "2016-05-08",

"Statement": [

{

“Sid”: “ListExistingFunctions”,

“Effect”: “Allow”,

“Action”: [

“lambda:ListFunctions”

],

“Resource”: “*”

}

]

}

We now need to enable the user to be able to view the details of the Lambda functions. Some of the information regarding the Lambda function include versions, aliases, and other information regarding con figurations. However, for this to be the case, the user has to be granted permission on the Lambda actions given below:

“Version”: “2016-05-08”,

“Statement”: [

{

“Sid”: “DisplayFunctionDetailsPermissions”,

“Effect”: “Allow”,

“Action”: [

“lambda:VersionsByFunctionList”,

“lambda:AliasesList”,

“lambda:GetFunction”,

“lambda:GetFunctionConfiguration”,

“lambda:EventSourceMappingsList”,

“lambda:GetPolicy”

],

“Resource”: “*”

}

]

}

We can then allow the user to invoke a Lambda function, and we need this to be done manually. For this to be the case, the user has to be granted permission on the action “*lambda:InvokeFunction.*” This can be done as shown below:

{

“Version”: “2016-05-06”,

“Statement”: [

{

“Sid”: “InvokePermission”,

“Effect”: “Allow”,

“Action”: [

“lambda:InvokeFunction”

],

“Resource”: “*”

}

]

}

A user can also be allowed to monitor a Lambda function and then view the CloudWatch Logs. After invocation of a Lambda function by a user, the AWS Lambda executes the function and then returns a result. For the user to be able to monitor the Lambda function, they need to be granted extra permissions.

For us to enable the user to be able to monitor the CloudWatch metrics for our Lambda function on the monitoring tab for our console, or on the grid view for our console home page, the following permissions have to be granted:

```
{
    “Version”: “2016-05-08”,
    “Statement”: [
    {
    “Sid”: “CloudWatchPermission”,
    “Effect”: “Allow”,
    “Action”: [
    “cloudwatch:GetMetricStatistics”
    ],
    “Resource”: “*”
    }
    ]
}
```

If you need your user to be in a position to click the links to the CloudWatch Logs in AWS Lambda and then view the log output in the CloudWatch logs, the following permissions have to be granted:

{

“Version”: “2016-05-08”,

“Statement”: [

{

“Sid”: “CloudWatchLogsPerms”,

“Effect”: “Allow”,

“Action”: [

“cloudwatchlog:DescribeLogGroups”,

“cloudwatchlog:DescribeLogStreams”,

“cloudwatchlog:GetLogEvents”

],

“Resource”: “arn:aws:logs:*region*:*account-id*:log-group:/aws/lambda/*”

}

]

}

It is also good for you to allow the user to be in a position to create a Lambda Function. However, for this to be the case, the following permissions have to be granted. The permissions which are related to IAM actions are needed since after creation of the Lambda function by the user, the user has to select the IAM execution role, and this is

what the AWS Lambda will assume so as to execute your Lambda function. This is shown below:

```
{
    “Version”: “2016-05-08”,
    “Statement”: [
    {
    “Sid”: “ExistingRolesAndPoliciesList”,
    “Effect”: “Allow”,
    “Action”: [
    “iam:RolePoliciesList”,
    “iam:RolesList”
    ],
    “Resource”: “*”
    },
    {
    “Sid”: “CreateFunctionPermissions”,
    “Effect”: “Allow”,
    “Action”: [
    “lambda:CreateFunction”
```

],

“Resource”: “*”

},

{

“Sid”: “PermissionToPassRole”,

“Effect”: “Allow”,

“Action”: [

“iam:PassRole”

],

“Resource”: “arn:aws:iam::*account-id*:role/*”

}

]

}

For the user to be in a position to create an IAM role once the user has created a Lambda function, the user has to be granted permission on the action “*iam:PutRolePolicy.*” The example given below shows how this happens:

{

“Sid”: “CreateRole”,

“Effect”: “Allow”,

“Action”: [

“iam:PutRolePolicy”

],

“Resource”: “arn:aws:iam::*account-id*:role/*”

}

Chapter 5- Permissions Required for Using AWS Lambda Console

For one to enjoy the benefits offered by the AWS Lambda console, they must be granted some extra permissions rather than the API-specific ones, and this is determined by what you expect your user to be in a position to do.

We need to discuss these permissions for the different integration points.

Amazon API Gateway

After configuring an API endpoint in the AWS Lambda console, the console always has to make several calls to the Gateway API. For these calls to work, the user has to be granted permission for the action "*apigateway:**." The example given below best demonstrates this:

{

"Version": "2016-05-08",

"Statement": [

{

"Sid": "ApiGatewayPermissions",

"Effect": "Allow",

"Action": [

"apigateway:*"

],

"Resource": "*"

},

{

“Sid”: “AddPermissionToFunctionPolicy”,

“Effect”: “Allow”,

“Action”: [

“lambda:AddPermission”,

“lambda:RemovePermission”,

“lambda:GetPolicy”

],

“Resource”: “arn:aws:lambda:*region*:*account-id*:function:*”

},

{

“Sid”: “ListEventSourcePerm”,

“Effect”: “Allow”,

“Action”: [

“lambda:ListEventSourceMappings”

],

“Resource”: “*”

}

]

}

Amazon CloudWatch Events

It is possible for you to schedule the time of execution of your Lambda function. After selecting a CloudWatch Events rule, AWS Lambda will create a new target in the CloudWatch which invokes the Lambda function. For the target creation to function, then the following permissions have to be granted:

{

“Version”: “2016-05-08”,

“Statement”: [

{

“Sid”: “EventPermissions”,

“Effect”: “Allow”,

“Action”: [

“events:PutRule”,

“events:ListRules”,

“events:ListRuleNamesByTarget”,

“events:PutTargets”,

“events:RemoveTargets”,

“events:DescribeRule”,

“events:TestEventPattern”,

“events:ListTargetsByRule”,

“events:DeleteRule”

],

“Resource”: “arn:aws:events:*region*:*account-id*:*”

},

{

“Sid”: “PermissionToFunctionPolicy”,

“Effect”: “Allow”,

“Action”: [

“lambda:AddPermission”,

“lambda:RemovePermission”,

“lambda:GetPolicy”

],

“Resource”: “arn:aws:lambda:*region*:*account-id*:function:*”

}

]

}

Amazon CloudWatch Logs

It is possible for you to invoke a Lambda function even when you have published events for Amazon CloudWatch Logs service. Once this service has been configured as an event source, the console will list the log groups contained in your AWS account. For the listing to take place, the permissions “logs:DescribeLogGroups” has to be granted. This is shown in the example given below:

```
{
    “Version”: “2016-05-08”,
    “Statement”: [
    {
    “Sid”: “CloudWatchLogsPermissions”,
    “Effect”: “Allow”,
    “Action”: [
    “logs:FilterLogEvents”,
    “logs:DescribeLogGroups”,
    “logs:PutSubscriptionFilter”,
    “logs:DescribeSubscriptionFilters”,
```

“logs:DeleteSubscriptionFilter”,

“logs:TestMetricFilter”

],

“Resource”: “arn:aws:logs:*region*:*account-id*:*”

},

{

“Sid”: “PermissionToFunctionPolicy”,

“Effect”: “Allow”,

“Action”: [

“lambda:AddPermission”,

“lambda:RemovePermission”,

“lambda:GetPolicy”

],

“Resource”: “arn:aws:lambda:*region*:*account-id*:function:*”

},

{

“Sid”: “EventSourceMappingsPermissions”,

“Effect”: “Allow”,

“Action”: [

“lambda:ListEventSourceMappings”

],

“Resource”: “*”

}

]

}

Amazon Cognito

The console is responsible for listing the identity pools contained in your account. Once you have selected the pool, it can be configured so that the event source type can be of the type “Cognito sync trigger.” For this to be done, the following permissions can be added:

{

“Version”: “2016-05-08”,

“Statement”: [

{

“Sid”: “CognitoPermissions1”,

“Effect”: “Allow”,

“Action”: [

“cognito-identity:ListIdentityPools”

],

“Resource”: [

“arn:aws:cognito-identity:region:account-id:*”

]

},

{

“Sid”: “CognitoPermissions2”,

“Effect”: “Allow”,

“Action”: [

“cognito-sync:GetCognitoEvents”,

“cognito-sync:SetCognitoEvents”

],

“Resource”: [

“arn:aws:cognito-sync:region:account-id:*”

]

},

{

“Sid”: “PermissionToFunctionPolicy”,

“Effect”: “Allow”,

“Action”: [

“lambda:AddPermission”,

“lambda:RemovePermission”,

“lambda:GetPolicy”

],

“Resource”: “arn:aws:lambda:region:account-id:function:*”

},

{

“Sid”: “EventSourcePerms”,

“Effect”: “Allow”,

“Action”: [

“lambda:ListEventSourceMappings”

],

“Resource”: “*”

}

]

}

Amazon DynamoDB

The console is good for listing the tables contained in your account. Once you have selected a table, the console will check to see if the DynamoDB stream is in existence for that specific table. If this is not found, the stream will be created. For the user to be allowed to create a DynamoDB stream to be an event source for the lambda function, the following additional permissions will have to be granted:

{

“Version”: “2016-05-08”,

“Statement”: [

{

“Sid”: “DynamoDBpermissions1”,

“Effect”: “Allow”,

“Action”: [

“dynamodb:DescribeStream”,

“dynamodb:DescribeTable”,

“dynamodb:UpdateTable”

],

“Resource”: “arn:aws:dynamodb:*region*:*account-id*:table/*”

},

{

“Sid”: “DynamoDBpermissions2”,

“Effect”: “Allow”,

“Action”: [

“dynamodb:ListStreams”,

“dynamodb:ListTables”

],

“Resource”: “*”

},

{

“Sid”: “LambdaGetPolicyPerm”,

“Effect”: “Allow”,

“Action”: [

“lambda:GetPolicy”

],

“Resource”: “arn:aws:lambda:*region*:*account-id*:function:*”

},

{

“Sid”: “LambdaEventSourcePerms”,

“Effect”: “Allow”,

“Action”: [

“lambda:CreateEventSourceMapping”,

“lambda:DeleteEventSourceMapping”,

“lambda:GetEventSourceMapping”,

“lambda:ListEventSourceMappings”,

“lambda:UpdateEventSourceMapping”

],

“Resource”: “*”

}

]

}

Amazon Kinesis Streams

The console will list all of the Amazon kinesis streams found in your account. Once a stream has been selected, the console will create event source mappings in your AWS Lambda. The following permissions have to be added so that this can work:

{

“Version”: “2016-05-08”,

“Statement”: [

{

“Sid”: “PermForDescStream”,

“Effect”: “Allow”,

“Action”: [

“kinesis:DescribeStream”

],

“Resource”: “arn:aws:kinesis:*region*:*account-id*:stream/*”

},

{

“Sid”: “PermForListStreams”,

“Effect”: “Allow”,

“Action”: [

“kinesis:ListStreams”

],

“Resource”: “*”

},

{

“Sid”: “PermForGetFunctionPolicy”,

“Effect”: “Allow”,

“Action”: [

“lambda:GetPolicy”

],

“Resource”: “arn:aws:lambda:*region*:*account-id*:function:*”

},

{

“Sid”: “LambdaEventSourcePermissions”,

“Effect”: “Allow”,

“Action”: [

“lambda:CreateEventSourceMapping”,

“lambda:DeleteEventSourceMapping”,

“lambda:GetEventSourceMapping”,

“lambda:ListEventSourceMappings”,

“lambda:UpdateEventSourceMapping”

],

“Resource”: “*”

}

]

}

Amazon S3

The console will work to prepopulate the list of the buckets contained in the AWS account and then find the bucket location for each of the buckets. Once Amazon S3 has been configured as an event source, the console will update the configuration of the bucket notification. For this to happen, the following permissions have to be added:

{

“Version”: “2016-05-08”,

“Statement”: [

{

“Sid”: “S3Permissions”,

“Effect”: “Allow”,

“Action”: [

“s3:GetBucketLocation”,

“s3:GetBucketNotification”,

“s3:PutBucketNotification”,

“s3:ListAllMyBuckets”

],

“Resource”: “arn:aws:s3:::*”

},

{

“Sid”: “PermissionToFunctionPolicy”,

“Effect”: “Allow”,

“Action”: [

“lambda:AddPermission”,

“lambda:RemovePermission”

],

“Resource”: “arn:aws:lambda:*region*:*account-id*:function:*”

}

]

}

Amazon SNS

The console is responsible for listing Amazon Simple Notification Service (SNS) topics in the account. Once you have selected a topic, the AWS Lambda will subscribe the Lambda function to the Amazon SNS topic. For this to work, the following permissions have to be added:

{

“Version”: “2016-05-08”,

“Statement”: [

{

“Sid”: “SNSPermissions”,

“Effect”: “Allow”,

“Action”: [

“sns:ListSubscriptions”,

“sns:ListSubscriptionsByTopic”,

“sns:ListTopics”,

“sns:Subscribe”,

“sns:Unsubscribe”

],

“Resource”: “arn:aws:sns:*region*:*account-id*:*”

},

{

“Sid”: “PermissionToFunctionPolicy”,

“Effect”: “Allow”,

“Action”: [

“lambda:AddPermission”,

“lambda:RemovePermission”,

“lambda:GetPolicy”

],

“Resource”: “arn:aws:lambda:*region*:*account-id*:function:*”

},

{

“Sid”: “LambdaESMappingsPermissions”,

“Effect”: “Allow”,

“Action”: [

“lambda:ListEventSourceMappings”

],

“Resource”: “*”

}

```
    ]
}
```

AWS IoT

The console is responsible for listing the available AWS IoT. Once you have selected a particular rule, the console will populate the rest of your information associated with the rule in the user interface. If the existing rule is selected, the console will update it with information so that the events will be send to the AWS Lambda. The following permissions have to be added for this to work:

{

“Version”: “2016-05-08”,

“Statement”: [

{

“Sid”: “IoTpermissions”,

“Effect”: “Allow”,

“Action”: [

“iot:GetTopicRule”,

“iot:CreateTopicRule”,

“iot:ReplaceTopicRule”

],

“Resource”: “arn:aws:iot:*region*:*account-id*:*”

},

{

“Sid”: “IoTTopicRulePermissions”,

“Effect”: “Allow”,

“Action”: [

“iot:ListTopicRules”

],

“Resource”: “*”

},

{

“Sid”: “LambdaPermissions”,

“Effect”: “Allow”,

“Action”: [

“lambda:AddPermission”,

“lambda:RemovePermission”,

“lambda:GetPolicy”

],

“Resource”: “arn:aws:lambda:*region*:*account-id*:function:*”

}

]

}

Chapter 6- Resource-Based Policies in AWS Lambda

A Lambda function is just one of the available resources in AWS Lambda. Permissions can be added to the policy associated with a particular Lambda function. Permission policies which have been attached to Lambda functions are usually known as *"resource-based policies."* Lambda function policies are used for the purpose of managing permissions for Lambda function invocation. Lambda function policies are used for management of permissions for Lambda function invocation.

The console will not directly support the modification of permissions directly in a function policy. One must use either AWS SDKs or AWS CLI. In this chapter, we are going to discuss the available AWS CLI operations.

Allow invocation of a Lambda Function by Amazon S3

For us to configure the Amazon S3 to be in a position to invoke the Lambda function, the permissions have to be configured as shown below:

- s3.amazonaws.com should be specified as the principal value.

- lambda:InvokeFunction should be specified as the action for which we are granting permissions.

If you need to f ensure that the event is to be generated from a specific bucket owned by a particular AWS account, the following should also be specified:

- The bucket ARN should be specified as the value for "*source-arn*" for restricting events from a particular bucket.

- Specify the ID of the AWS account which owns the bucket, so as to make sure that the bucket you name is owned by the bucket.

Consider the AWS CLI command given below, which adds permission to the Lambda function policy named "*hellothere,*" and it will grant permission to the Amazon S3 to invoke the Lambda function:

**aws lambda add-permission **

**—region asia-west-2 **

**—function-name hellothere **

**—statement-id 1 **

**—principal s3.amazonaws.com **

**—action lambda:InvokeFunction **

**—source-arn arn:aws:s3:::samplebucket **

**—source-account 111111111111 **

—profile adminuser

In the above example, it has been assumed that the "*adminuser*" is the one adding the

permission. This means that the parameter *"—profile"* has been used for specifying the profile of the adminuser.

In response to the above code, the AWS Lambda will return some JSON code. The value "*statement"* is just a string version in JSON for the statement which has been added to our Lambda function policy.

{

"Statement": "{\"Condition\":{\"StringEquals\": {\"AWS:SourceAccount\":\"12222287611\"},

\"ArnLike\":{\"AWS:SourceArn\":\"arn:aws:s3:::samplebucket\"}},

\"Action\":[\"lambda:InvokeFunction\"],

\"Resource\":\"arn:aws:lambda:asia-west-2: 12222287611:function:hellothere\",

\"Effect\":\"Allow\",\"Principal\":{\"Service\":\"s3.amazonaws.com\"},

\"Sid\":\"1\"}"

}

Permitting Amazon API Gateway to call a Lambda Function

For the Amazon APAI gateway to be allowed to be able to invoke a Lambda function, we should do the following:

- The “apigateway.amazonaws.com” should be specified as the principal value.

- The “lambda:InvokeFunction” should then be specified as the action to be used for granting permissions.

- API Gateway endpoint ARN should be specified as the “*source-arn”* value.

The AWS CLI command given below will add permission to the Lambda function policy named “*hellothere”* so as to grant permissions to the API gateway to be able to invoke the Lambda function. Here it is:

**aws lambda add-permission **

**—region asia-west-2 **

**—function-name hellothere **

**—statement-id 5 **

**—principal apigateway.amazonaws.com **

**—action lambda:InvokeFunction **

**—source-arn arn:aws:execute-api:*region*:*account-id*:*api-id*/*stage*/*method*/resource-path" **

—profile adminuser

The AWS Lambda will respond to the above code with a JSON code. The value "*statement*" is just a JSON string version for the statement which had been added to our Lambda function policy.

{

"Statement": "{\"Condition\":{\"ArnLike\":{\"AWS:SourceArn\":\"arn:aws:apigateway:asia-east-1::my-api-id:/test/walkthrough/dogs\"}},

\"Action\":[\"lambda:InvokeFunction\"],

\"Resource\":\"arn:aws:lambda:asia-west-2:account-id:function:helloworld\",

\"Effect\":\"Allow\",

\"Principal\":{\"Service\":\"apigateway.amazonaws.com\"},

\“Sid\”:\“5\”}”

}

A user application, which has been created by another AWS account can also be allowed to invoke a Lambda function, with a cross-account scenario. However, for this to be the case, the account ID for the AWS account should be specified as the “principal.” The AWS CLI command given below best demonstrates this:

**aws lambda add-permission **

**—region asia-west-2 **

**—function-name hellothere **

**—statement-id 3 **

**—principal 12222287611 **

**—action lambda:InvokeFunction **

—profile adminuser

The AWS Lambda will respond by giving the following code. The value “statement” is the JSON string version of statement which has been added to our Lambda function policy. Here is the response:

{

“Statement”: “{\“Action\”:[\“lambda:InvokeFunction\”],

\“Resource\”:\“arn:aws:lambda:asia-west-2:account-id:function:hellothere\”,

\“Effect\”:\“Allow\”,

\“Principal\”:{\“AWS\”:\“account-id\”},

\“Sid\”:\“3\”}”

}

Retrieving a Lambda Function Policy

For you to retrieve the Lambda function policy, you should use the command "*get-policy.*" This is shown below:

**aws lambda get-policy **

**—function-name sample **

—profile adminuser

Removing permissions from the Lambda Function Policy

This is possible, and it is done using the command "*remove-policy.*" However, you have to specify the name of the function and the statement ID. This is best demonstrated in the example given below:

**aws lambda remove-permission **

**—function-name sample **

**—statement-id 1 **

—profile adminuser

Chapter 7- Actions

Let us discuss some of the actions which are supported in AWS Lambda.

CreateAlias

This is used for creating an alias to the function which points to a specified version of a Lambda function. Alias names are always unique to the given function. For this to work, the permission for the action “lambda:CreateAlias” has to be added.

It takes the following as the request syntax:

POST /2016-05-31/functions/*FunctionName*/aliases HTTP/1.1

Content-type: application/json

{

“Description”: “*string*“,

“FunctionVersion”: “*string*“,

“Name”: “*string*”

}

The following are the required URI parameters:

FunctionName- this represents the name of your Lambda function for which an alias is to be created. It should have a minimum length of 1, and a maximum length of 140.

The request should have the following data specified in JSON format:

- Description- this should be of the type string and describes the alias to be created.

- FunctionVersion- this is the version of the function for which you are to create an alias.

The response syntax should be as follows:

HTTP/1.1 201

Content-type: application/json

{

“AliasArn”: “*string*“,

“Description”: “*string*“,

“FunctionVersion”: “*string*“,

“Name”: “*string*”

}

DeleteAlias

This is used for deleting the Lambda function alias that you specify. For this to happen, you have to specify the permission for the action “lambda:DeleteAlias.”

The following should be the request syntax:

DELETE /2016-05-09/functions/FunctionName/aliases/Name HTTP/1.1

The following URI parameters are required in the request:

- FunctionName- this is the name of the function for which an alias is to be created.

- Name- this is the name for the alias which you are intending to delete.

You should then get the following as the response syntax:

HTTP/1.1 204

Conclusion

We have come to the conclusion of this guide. My hope is that you have learned most of the properties associated with AWS Lambda. You should now be aware of how to create a deployment package. We have discussed how to do this in Node, Java, and Python. In Node, you can choose to do it by writing the code on your own or by using the console. Note that the deployment package can be created in either a .jar or .zip form, so you can choose how to do this.

In Java, one can choose to use Maven for the purpose of creating the deployment package. Again, the package can either be in a .zip or .jar form. You can choose to use the Maven package alone, in which case you will first begin by installing it. Also, you can choose to combine the Maven with Eclipse IDE for Java, in which case you will have to install the Maven plugin for the Eclipse. For you to be able to carry out certain tasks in AWS Lambda, there are some permissions and actions which have to be permitted in a particular account, and this has been discussed.

Limits to the usage of resources are alterable, and it is good for you to know how to do this so as to take care of concurrent executions.