AWS Lambda

Developer Guide

AWS Lambda: Developer Guide

Table of Contents

What Is AWS Lambda?

AWS Lambda makes it easy for you to build applications that respond quickly to new information. You upload your application code as "Lambda" functions and AWS Lambda runs your code on high-availability compute infrastructure and performs all the administration of the compute resources, including server and operating system maintenance, capacity provisioning and automatic scaling, code and security patch deployment, and code monitoring and logging. All you need to do is supply your code in one of the languages that AWS Lambda supports (currently Node.js or Java).

AWS Lambda can execute your Lambda functions in response to one of the following:

- Events, such as discrete updates (for example, object-created events in Amazon S3 or CloudWatch alerts), or streaming updates (for example, website clickstreams or outputs from connected devices).
- JSON inputs or HTTPS commands from your custom applications.

AWS Lambda executes your code only when needed and scales automatically, from a few requests per day to thousands per second. With these capabilities, you can use Lambda to easily build triggers for AWS services like Amazon S3 and Amazon DynamoDB, process streaming data stored in Amazon Kinesis, or create your own back-end that operates at AWS scale, performance, and security.

When should I use AWS Lambda?

Amazon Web Services lets you choose from a range of compute services to meet your needs.

- The Amazon Elastic Compute Cloud (Amazon EC2) web service offers flexibility and a wide range of EC2 instance types to choose from. It gives you the option to customize operating systems, network and security settings, and the entire software stack, but you are responsible for provisioning capacity, monitoring fleet health and performance, and using Availability Zones for fault tolerance.
- Elastic Beanstalk offers an easy-to-use service for deploying and scaling applications onto Amazon EC2 in which you retain ownership and full control over the underlying EC2 instances.

AWS Lambda is a great alternative to using these other AWS compute services if you can write your application code in languages supported by AWS Lambda, and run within the standard runtime environment and resources provided by the service.

The higher-level abstraction that AWS Lambda offers is the convenience of your being responsible only for your code. AWS Lambda manages the compute fleet that offers a balance of memory, CPU, network

and other resources. This is in exchange for flexibility, which means you cannot log in to compute instances, customize the operating system or language runtime. These constraints enable AWS Lambda to perform operational and administrative activities on your behalf, including provisioning capacity, monitoring fleet health, applying security patches, deploying your code, running a web service front end, and monitoring and logging your functions.

Are you a first-time user of AWS Lambda?

If you are a first-time user of AWS Lambda, we recommend you read the following sections in order:

1. For a product overview and pricing information, go to the AWS Lambda product detail page.
2. Read AWS Lambda: How it Works (p. 3).
3. Depending on the language (Java or Node.js) you want to author your Lambda function in, follow one of the getting started exercises:
   - Getting Started: Authoring AWS Lambda Code in Node.js (p. 15)
   - Getting Started: Authoring AWS Lambda Code in Java (p. 88)
4. To further explore authoring AWS Lambda functions, see the following topics:
   - Authoring Lambda Functions in Node.js (p. 15)
   - Authoring Lambda Functions in Java (p. 88)

Refer to the following topics to learn more about the service:

- Troubleshooting and Monitoring AWS Lambda Functions with Amazon CloudWatch (p. 135)
- Best Practices for Working with AWS Lambda Functions (p. 145)
- AWS Lambda Limits (p. 146)

AWS Lambda: How it Works

The following sections provide an overview of various AWS Lambda service components and how they interact.

> **Note**
> After you read the introduction, try the Getting Started exercises and the walkthroughs for Node.js and Java.

Topics

Core Components: Lambda Function and Event Source

A Lambda function and an event source are the core components when you work with AWS Lambda. Event sources publish events, and a Lambda function is the custom code that you write to process the events. You can also invoke your Lambda function over HTTPS.

Then, AWS Lambda executes your Lambda function on your behalf.

Lambda Function

After you upload your custom code to AWS Lambda, we refer to it as a Lambda function. A Lambda function consists of your code, associated dependencies, and configuration. Configuration includes information such as the `handler` that will receive the event, the IAM role AWS Lambda can assume to execute the Lambda function on your behalf, the compute resource you want allocated and execution timeout.

Currently, you can author your Lambda function code in Java or Node.js.

Event Source

Event sources publish events that cause the Lambda function to be invoked (either by the event source in the "push" model, or by AWS Lambda in the "pull" model, discussed in the next section). Event sources can be:

- AWS services such as Amazon S3.
- Amazon services such as Amazon Echo.
- User-defined applications such as your Android application.

All of these invoke Lambda functions using the Lambda `invoke` API.

AWS services—Currently, AWS Lambda supports events from Amazon S3, Amazon DynamoDB, Amazon Kinesis, Amazon SNS, and Amazon Cognito.

You can also use Lambda functions with other AWS services that publish data to one of these listed event sources. For example:

- You can trigger Lambda functions in response to CloudTrail updates because it records all API access events to an S3 bucket.
- You can trigger Lambda functions in response to Amazon CloudWatch alarms because it publishes alarm events to an Amazon SNS topic.

Each of these event sources have a pre-defined event data structure and you can write a Lambda function to process the event data that it receives. For example, Amazon S3 can publish object-created events, Amazon DynamoDB can publish table updates to a stream associated with the table, and AWS CloudTrail can record all API calls made in your account.

Other Amazon Services—Currently, you can use Lambda functions to build services that give new “skills” to Alexa, the Voice assistant on Amazon Echo. The Alexa Skills Kit provides the APIs, tools, and documentation to create these new skills, powered by your own services running as Lambda functions. Amazon Echo users can access these new skills by asking Alexa questions or making requests. For more information, go to Getting Started with Alexa Skills Kit.

Build your own event source—In addition to AWS services generating events, user applications can also generate events. User applications such as client, mobile, or web applications can publish events and invoke Lambda functions using the AWS SDKs or AWS Mobile SDKs.

Invoking Lambda Functions Over HTTPS

In addition to invoking Lambda functions using event sources or through custom clients that are built using the AWS SDKs, you can also invoke your Lambda function over HTTPS. You can do this by defining a custom REST API and endpoint using Amazon API Gateway. You map individual API methods, such as GET and PUT, to specific Lambda functions. When you send an HTTPS request to the API endpoint, the Amazon API Gateway service invokes the corresponding Lambda function.

For more information, see http://aws.amazon.com/api-gateway/.

Invocation Types

AWS Lambda supports the following two invocation types:

- `Event` invocation type:

This invocation type causes AWS Lambda to execute the Lambda function asynchronously. The event sources Amazon S3, Amazon SNS, Amazon Kinesis, or Amazon Cognito use this invocation type. Your custom applications can also invoke your Lambda function using this invocation type.

Because of the asynchronous nature of execution, your Lambda function need not send any response.

- `RequestResponse` invocation type:

  This invocation type causes AWS Lambda to execute the function synchronously and returns the response immediately to the calling application. This invocation type is available only for custom applications.

  For more information, see Invoke (p. 178).

  Note
  When you invoke a Lambda function via the AWS console or over HTTPS using Amazon API Gateway, Lambda always uses the `RequestResponse` invocation type.

Related Topics

AWS Lambda: How it Works (p. 3)

The Pull/Push Event Models

How a Lambda function is invoked depends on the event source that you use with it. There are two models:

- The *pull event* model
- The *push event* model

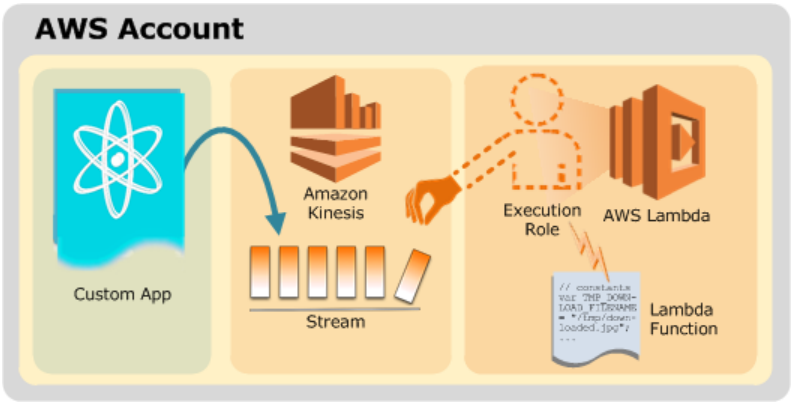
Note
This section applies when using a Lambda function with event sources and does not apply when you are invoking over HTTPS using Amazon API Gateway.

The Pull Event Model

In the pull event model, AWS Lambda polls the event source and invokes your Lambda function when it detects an event. This model applies when AWS Lambda is used with streaming event sources such as Amazon Kinesis and DynamoDB Streams. For example, AWS Lambda polls your Amazon Kinesis stream, or Amazon DynamoDB Stream, and invokes your Lambda function when it detects new records on the stream.

In this model, AWS Lambda manages the event source mapping. That is, it provides an API for you to create event source mappings that associate your Lambda function with a specific event source. For more information, see CreateEventSourceMapping (p. 156). You will need to grant AWS Lambda necessary permissions to access the event source via an IAM role. The permission model is discussed in the next section.

For example, the following diagram shows AWS Lambda polling an Amazon Kinesis stream and invoking the Lambda function when it detects new events.



The Push Event Model

In push event model, an event source directly invokes a Lambda function when it publishes an event. The push model applies to Amazon S3, Amazon SNS, Amazon Cognito, Amazon Echo and user applications, where each individual event triggers the Lambda function. You will need to grant the event source necessary permissions to invoke the Lambda function via the access policy associated with the function. For more information about the permission model, see Permission Model (p. 8).

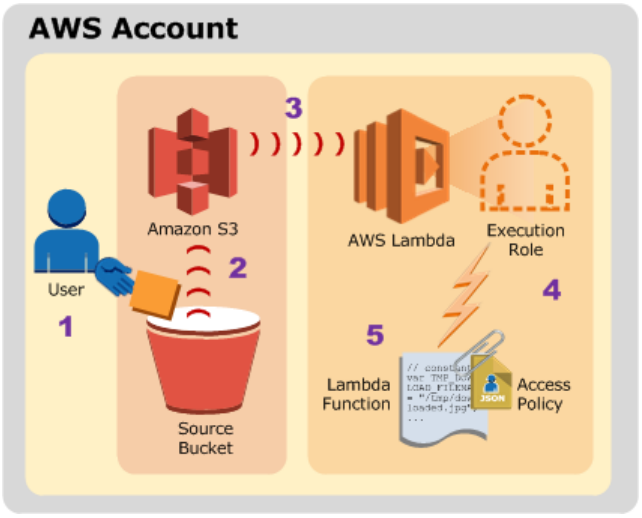
AWS Lambda does not explicitly manage event sources in this model. That is, you do not use the AWS Lambda APIs to map your Lambda function to its event source. Instead, you use APIs from the event source to configure this mapping. For example, if Amazon S3 is your event source, you would specify event source mapping as part of bucket notification configuration.

> **Note**
> For your convenience, the AWS Lambda console shows all event sources (pull and push) for a given function.

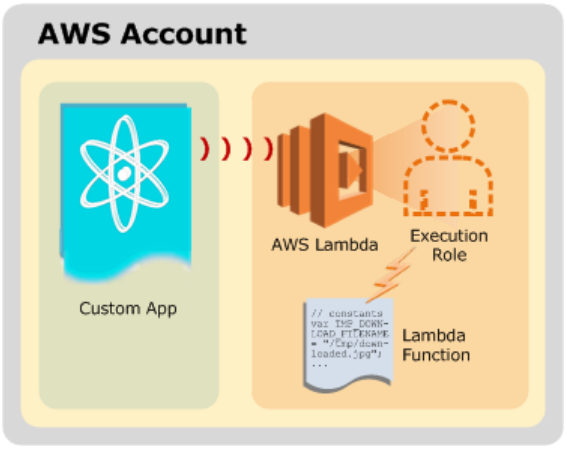
The event source can decide the invocation type it wants to use. For more information, see Invoke (p. 178). Amazon S3, Amazon SNS, Amazon Cognito, and Amazon Echo invoke the Lambda function using the "event" invocation type in the invoke API call. User applications can invoke the Lambda functions by using the "`Event`" invocation type or the "`RequestResponse`" invocation type.

The following are examples of the push event model:

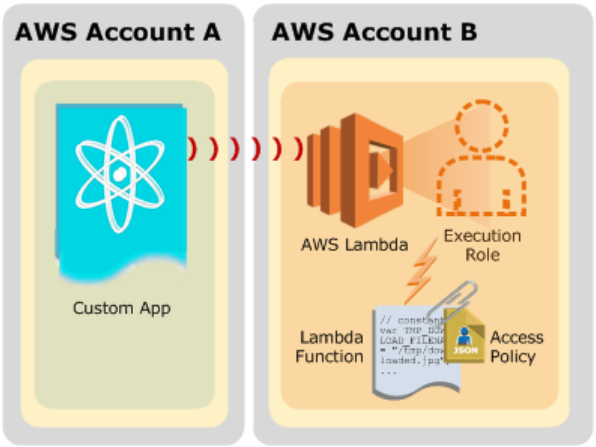
- The following diagram shows Amazon S3 invoking a Lambda function upon detecting an object-created event in a bucket. The Lambda function owner must grant Amazon S3 permission to invoke the function by adding a permission in the access policy associated with the Lambda function. Access policies are discussed in Permission Model (p. 8).



- The following diagram shows a custom application in your account invoking your Lambda function.



- In the following diagram, the user application and Lambda function are owned by different AWS accounts. In this case, the Lambda function owner must add a permission in the access policy associated with the Lambda function to grant cross-account access. Cross accounts are discussed in Permission Model (p. 8).



Related Topics

AWS Lambda: How it Works (p. 3)

Permission Model

There are two types of permissions related to Lambda functions:

- **Execution permissions**—The permissions that your Lambda function needs to access other AWS resources in your account. You grant these permissions by creating an IAM role, known as an *execution role*.
- **Invocation permissions**—The permissions that the event source needs to communicate with your Lambda function. Depending on the invocation model (push or pull model), you can grant these permissions using either the execution role or resource policies (the access policy associated with your Lambda function).

Execution Permissions

When your Lambda function executes, it can access other AWS resources in your account. For example:

- Read objects from an S3 bucket and write objects to an S3 bucket
- Write logs to CloudWatch Logs.
- Write items to an Amazon DynamoDB table.

You must grant necessary permissions for the Lambda function to access these resources.

You grant these permissions via an IAM role, called execution role. You create the IAM role granting AWS Lambda permission to assume the role and attach an access policy to the role granting all the needed permissions.

First you create an execution role, and then you specify the role at the time you create your Lambda function so that AWS Lambda knows the role it can assume to execute your Lambda function on your behalf.

Each IAM role has two policies attached:

- **Access policy**—Grants your Lambda function the resource permissions it needs.

  For example, if your Lambda function writes logs for your function to Amazon CloudWatch Logs, you grant permissions for the CloudWatch Logs actions to create a log group, log stream, and write to the log stream as shown:

```
{
    "Statement": [
        {
            "Action": [
                "logs:CreateLogGroup",
                 "logs:CreateLogStream",
                 "logs:PutLogEvents"
            ],
            "Effect": "Allow",
            "Resource": "arn:aws:logs:*:*:*"
```

```
        }
    ]
}
```

If your function uploads an object to an S3 bucket, you grant permission for relevant Amazon S3 actions in this policy. For a list of actions, see Specifying Permissions in a Policy in the *Amazon Simple Storage Service Developer Guide*.

- **Trust policy**—Identifies who can assume the role. In this case, it is the AWS Lambda service principal as shown:

  Important
  The user who is creating the IAM role is passing permission to AWS Lambda to assume this role. This requires the user to have permission for the `iam:PassRole` action to be able to grant this permission. If an administrator user is creating this role, the user has full permissions including the `iam:PassRole`.

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

For more information about IAM roles, go to Roles (Delegation and Federation) in IAM User Guide.

Invocation Permissions

The entity invoking your Lambda function must have permission to do so.

- In the *pull* model, AWS Lambda invokes your Lambda function but it needs to read your event sources (such as a Kinesis stream or a DynamoDB stream). Therefore you will need to grant AWS Lambda permission to read from the stream. You do this by updating the execution role associated with your Lambda function.
- In the *push* model, the event source (such as Amazon S3 or a user-defined application) invokes your Lambda function by publishing events. You grant these event sources permission to invoke your Lambda function by updating the access policy associated with your Lambda function. AWS Lambda provides the `AddPermission` API for this purpose. For more information, see AddPermission (p. 153).

  Note
  If the user-defined application and the Lambda function it invokes belong to the same AWS account, you don't need to grant explicit permissions.

The following are two example policies:

- Grant Amazon S3 permission to invoke your Lambda function—The following example policy grants Amazon S3 `Principal` permission for the `lambda:InvokeFunction` action provided that the event source is `examplebucket` bucket and the bucket is owned by a specific AWS account.

```
{
   "Statement":{
      "StatementId":"Id-1",
      "Action":"lambda:InvokeFunction",
      "Principal":"s3.amazonaws.com",
      "SourceArn":"arn:aws:s3:::examplebucket",
      "SourceAccount":"account-id"
   }
}
```

- Grant Amazon API Gateway permission to invoke your Lambda function—The following example policy grants Amazon API Gateway `Principal` permission for the `lambda:InvokeFunction` action provided that the request is coming from your Amazon API Gateway API resource action.

```
{
   "Statement":{
      "StatementId":"Id-1",
      "Action":"lambda:InvokeFunction",
      "Principal":"apigateway.amazonaws.com",
      "SourceArn":"  arn:aws:execute-api:aws-region:ac
count_id:api_id/stage/method/api_specific_resource_path",
      "SourceAccount":"account-id"
   }
}
```

- Grant cross-account permission to a user application created by some other AWS account—To grant cross-account permission to another AWS account, you specify `Principal` as shown:

```
"Principal": "account-id"
```

For more information, see Principal in *IAM User Guide*.

Related Topics

AWS Lambda: How it Works (p. 3)

Resource Model

In the AWS Lambda resource model, you choose the amount of memory you want allocated for your function. AWS Lambda then allocates CPU power proportional to the memory by using the same ratio as a general purpose Amazon EC2 instance type, such as an M3 type. For example, if you allocate 256 MB to your Lambda function, it will receive twice the CPU share than if you had allocated 128 MB. You can request additional memory in 64 MB increments up to 1536 MB. For a full list of AWS Lambda limits, see AWS Lambda Limits (p. 146).

Next Step

Set Up an AWS Account and Create an Administrator User (p. 12)

Supported Versions

The AWS Lambda runtime supports the following versions:

Item	Version
Public Amazon Linux AMI version	AMI Id: ami-e7527ed7 in the US West (Oregon) region. For information about using an AMI, see Amazon Machine Images (AMI) in the *Amazon EC2 User Guide for Linux Instances*.
Linux kernel version	3.14.44-32.39.amzn1.x86_64
Node.js	v0.10.36
ImageMagick	Installed with default settings. For versioning information, go to imagemagick nodejs wrapper and ImageMagick native binary (search for "ImageMagick").
AWS SDK for JavaScript version	AWS SDK for JavaScript version 2.1.35

Next Step

Set Up an AWS Account and Create an Administrator User (p. 12)

Set Up an AWS Account and Create an Administrator User

Before you use AWS Lambda for the first time, complete the following tasks:

1. Sign up for AWS (p. 12)
2. Create an IAM User (p. 12)

Sign up for AWS

When you sign up for Amazon Web Services (AWS), your AWS account is automatically signed up for all services in AWS, including AWS Lambda. You are charged only for the services that you use.

With AWS Lambda, you pay only for the resources you use. For more information about Amazon Lambda usage rates, see the AWS Lambda product page. If you are a new AWS customer, you can get started with AWS Lambda for free; for more information, see AWS Free Usage Tier.

If you have an AWS account already, skip to the next task. If you don't have an AWS account, use the following procedure to create one.

To create an AWS account

1. Open http://aws.amazon.com/, and then click **Sign Up**.
2. Follow the on-screen instructions.

   Part of the sign-up procedure involves receiving a phone call and entering a PIN using the phone keypad.

Note your AWS account ID, because you'll need it for the next task.

Create an IAM User

Services in AWS, such as AWS Lambda, require that you provide credentials when you access them, so that the service can determine whether you have permission to access its resources. The console requires

your password. You can create access keys for your AWS account to access the command line interface or API. However, we don't recommend that you access AWS using the credentials for your AWS account; we recommend that you use AWS Identity and Access Management (IAM) instead. Create an IAM user, and then add the user to an IAM group with administrative permissions or and grant this user administrative permissions. You can then access AWS using a special URL and the credentials for the IAM user.

If you signed up for AWS but have not created an IAM user for yourself, you can create one using the IAM console.

The getting started and walkthrough exercises in this guide assume you have a user ("adminuser") with administrator privileges. So when you follow the procedure, create a user with name "adminuser".

To create the Administrators group

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. In the navigation pane, click **Groups**, and then click **Create New Group**.
3. In the **Group Name** box, type **`Administrators`**, and then click **Next Step**.
4. In the list of policies, select the check box next to the **AdministratorAccess** policy. You can use the **Filter** menu and the **Search** box to filter the list of policies.
5. Click **Next Step**, and then click **Create Group**.

Your new group is listed under **Group Name**.

To create an IAM user for yourself, add the user to the Administrators group, and create a password for the user

1. In the navigation pane, click **Users**, and then click **Create New Users**.
2. In box **1**, type a user name. Clear the check box next to **Generate an access key for each user**. Then click **Create**.
3. In the list of users, click the name (not the check box) of the user you just created. You can use the **Search** box to search for the user name.
4. In the **Groups** section, click **Add User to Groups**.
5. Select the check box next to the **Administrators** group. Then click **Add to Groups**.
6. Scroll down to the **Security Credentials** section. Under **Sign-In Credentials**, click **Manage Password**.
7. Select **Assign a custom password**. Then type a password in the **Password** and **Confirm Password** boxes. When you are finished, click **Apply**.

To sign in as this new IAM user, do the following:

1. Sign out of the AWS Management Console.
2. Use the following URL

   ```
   https://aws_account_number.signin.aws.amazon.com/console/
   ```

   The *`aws_account_number`* is your AWS account ID without hyphen. For example, if your AWS account ID is `1234-5678-9012`, your AWS account number is `123456789012`.

   For information about finding your account number, go to Your AWS Account ID and Its Alias in the IAM User Guide.
3. Enter the IAM user name and password that you just created. When you're signed in, the navigation bar displays "*your_user_name @ your_aws_account_id*".

If you don't want the URL for your sign-in page to contain your AWS account ID, you can create an account alias.

To create or remove an account alias

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. On the navigation pane, select **Dashboard**.
3. Find the IAM users sign-in link.
4. To create the alias, click **Customize**, enter the name you want to use for your alias, and then click **Yes, Create**.
5. To remove the alias, click **Customize**, and then click **Yes, Delete**. The sign-in URL reverts to using your AWS account ID.

To sign in after you create an account alias, use the following URL:

```
https://your_account_alias.signin.aws.amazon.com/console/
```

To verify the sign-in link for IAM users for your account, open the IAM console and check under **IAM users sign-in link:** on the dashboard.

For more information about IAM, see the following:

- Identity and Access Management (IAM)
- IAM Getting Started Guide
- IAM User Guide

Next Step

Getting Started: Authoring AWS Lambda Code in Node.js (p. 15)

Authoring Lambda Functions in Node.js

This section explains how to author your Lambda functions in Node.js. We recommend you first review the AWS Lambda: How it Works (p. 3) section and make sure you are familiar with core AWS Lambda concepts such as function, event source, event source mapping, Lambda permission model, and resource model. You can then review the topics in this section for information specific to creating Lambda functions in Node.js.

Topics

Getting Started: Authoring AWS Lambda Code in Node.js

The Getting Started section provide exercises using the AWS Lambda console. Before you explore the Getting Started exercises, we recommend you read introductory information including the core components of AWS Lambda, the programming model, and the permissions model. For more information, see AWS Lambda: How it Works (p. 3). Then, you can review the following three Getting Started exercises.

Getting Started 1: Invoking Lambda Functions from User Applications Using the AWS Lambda Console (Node.js) (p. 16)

Getting Started 2: Handling Amazon S3 Events Using the AWS Lambda Console (Node.js) (p. 19)

Getting Started 3: Handling Amazon Kinesis Events Using the AWS Lambda Console (Node.js) (p. 23)

The AWS Lambda console provides a set of blueprints that you can use to easily create Lambda functions and test them. The getting started exercises use these blueprints.

Note that the console does many things for you as you create and configure Lambda functions. To help you understand the AWS Lambda API, the documentation also provides AWS CLI–based walkthroughs. For more information, see AWS Lambda Walkthroughs (Node.js) (p. 35).

Preparing for the Getting Started

First, you need to sign up for an AWS account and create an administrator user in your account. For instructions, see Set Up an AWS Account and Create an Administrator User (p. 12).

> **Important**
> AWS Identity and Access Management recommends that you do not use the root credentials of your AWS account to make requests. Instead, create an IAM user (called *adminuser*), grant that user full access, and then use that user's credentials to interact with AWS. We refer to this user as an administrator user. For more information, go to Root Account Credentials vs. IAM User Credentials in the *AWS General Reference* and IAM Best Practices in *Using IAM*.

Getting Started 1: Invoking Lambda Functions from User Applications Using the AWS Lambda Console (Node.js)

Topics

One of the use cases for using AWS Lambda is to process events generated by a user application. For demonstration purposes, you don't need to write a user application that will invoke your Lambda function. Instead, in this Getting Started exercise, you will use sample event data and invoke your Lambda function manually.

> **Note**
> This is an example of the AWS Lambda *request-response* model in which a user application invokes a Lambda function and receives a response in real time. For more information, see AWS Lambda: How it Works (p. 3).

You will do this exercise using the AWS Lambda console. You will use the console provided "hello-world" blueprint and event data.

- The "hello-world" blueprint assumes application generating event data of this form:

```
{
  "key1": "value1",
  "key2": "value2",
  "key3": "value3"
}
```

- Accordingly, the blueprint provides the following Node.js example code to process these events. You will use this sample code to create your Lambda function.

```
console.log('Loading function');

exports.handler = function(event, context) {
```

```
    console.log('value1 =', event.key1);
    console.log('value2 =', event.key2);
    console.log('value3 =', event.key3);
    context.succeed(event.key1);  // Echo back the first key value
    // context.fail('Something went wrong');
};
```

In the code, the `handler` receives the event as the first parameter. and processes the event, for illustration it simply writes the incoming event data to CloudWatch logs, each `console.log()` statements generate log events in CloudWatch.

Step 1: Create a Lambda Function

Follow the steps to create a Lambda function.

1. Sign in to the AWS Management Console and open the AWS Lambda console.
2. Choose **Get Started Now**.

   Note
   The console shows the **Get Started Now** page only if you do not have any Lambda functions created. If you have created functions already, you will see the **Lambda > Functions** page. On the list page, choose **Create a Lambda function** to go to the **Lambda > New function** page.

3. On the **Step 1: Select blueprint** page, select the "Hello-world" blueprint.
4. On the **Step 2: Configure function** page, do the following:

   a. In the **Configure function** section, enter the function name **`HelloWorld`** in the **Name** box.
   b. In the **Lambda Function Code** section, do the following:

      - Provide code for the Lambda function that you are creating.
      - Specify the IAM execution role that AWS Lambda can assume when it executes the function on your behalf. For more information, see Execution Permissions (p. 8). The example Lambda function only writes logs to Amazon CloudWatch. In this case, use the **Basic execution role** template that grants permissions for CloudWatch actions.

      i. Select Node.js from the **Runtime** list.
      ii. Choose **Edit code inline**. Review the code. You will upload this code as your Lambda function.
      iii. Note the value in the **Handler** box matches the `exports.handler` in the code.

When you create a Lambda function you must specify the `handler` in your code that will receive the events and process them. In the console you specify the handler in the `Handler` box. For more information, see Programming Model (Node.js) (p. 32).

iv. From the **Role** list, choose **Basic Execution Role** in the **Create New Role** section (you might need to enable pop-ups to see the role selector list).

When you create a Lambda function, you must specify an IAM role that AWS Lambda can assume to execute your Lambda function on your behalf. The role must have permissions for the AWS action that your Lambda function will need. The Lambda function in this example only writes logs to CloudWatch Logs. So the following predefined access policy associated with the basic execution role is sufficient for the Lambda function you are creating.

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

On the **AWS Lambda requires access to your resource** page, do the following:

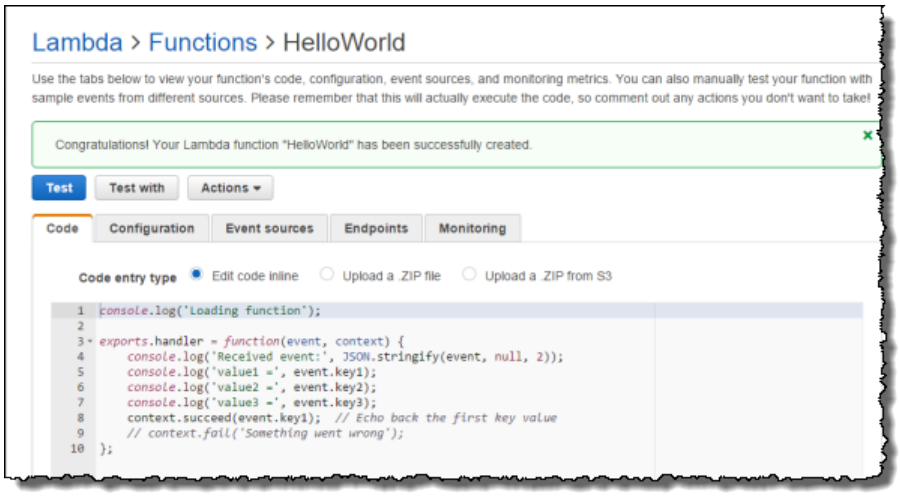
A. Choose **Create a new IAM Role** from the **IAM Role** list.
B. For **Role Name**, enter a role name. For example, type **`lambda_basic_execution`** for the execution role that you need to create for the first Getting Started exercise.
C. Choose **View Policy Document** to review the access policy.
D. Choose **Allow**.

The browser tab will close, and the new role name appears on the **Lambda: New Function** page.

c. In the **Advanced settings** section, leave the default Lambda function configuration values. The memory and timeout values are sufficient for the Lambda function you are creating.

d. Choose **Create Function** to create a Lambda function.

The console saves the code into a file and then zips the file, which is the deployment package. The console then uploads the deployment packages to AWS Lambda creating your Lambda function. The console shows the HelloWorld Lambda function, you can now perform various action including test the function:



Step 2: Invoke the Lambda Function Manually

Follow the steps to invoke your Lambda function using console-provided sample event data.

1. In the **> Functions > HelloWorld** page, click **Test with** to select sample event data that you want to pass to the Lambda function.
2. In the **Input sample event** page, select the "Hello World" from the **Sample event** list and click **Submit**.

   You can change key and values in the sample JSON but don't change the event structure.
3. AWS Lambda executes your function on your behalf (by assuming the execution role you had specified at the time of creating the function). The handler in your Lambda function receives the sample event and processes it.
4. View results in the console. Note the following:

   - The **Execution result** section shows the object passed to the `context.succeed()` method in your code.
   - The **Summary** section shows the REPORT line in the execution log.
   - The **Execution log** section shows the log AWS Lambda generates for each execution.

   Note that the **Click here** link shows logs in the CloudWatch console. The function then adds logs to Amazon CloudWatch, to the log group that corresponds to the Lambda function.

Next Step

Getting Started 2: Handling Amazon S3 Events Using the AWS Lambda Console (Node.js) (p. 19)

Getting Started 2: Handling Amazon S3 Events Using the AWS Lambda Console (Node.js)

In this Getting Started exercise, you create a Lambda function to consume events published by Amazon S3.

Amazon S3 notification feature enables you to configure notification on a bucket and request Amazon S3 to publish object-created events to AWS Lambda by invoking your Lambda function. AWS Lambda executes the function by passing the event data to the `handler` in your Lambda function. The handler then processes the event. For illustration, in this example the handler logs some of the event information to Amazon CloudWatch.

> **Note**
> This is an example of the "push" model where Amazon S3 invokes the Lambda function and then AWS Lambda executes it. For more information about the "push" model, see AWS Lambda: How it Works (p. 3).

You will perform tasks in this exercise in two steps:

- Create a Lambda function and manually invoke it using a sample Amazon S3 event.
- Add the event source. This is where you configure your Lambda function to respond to events published by Amazon S3.

  In a "push" model, event source mapping is managed by the individual service. In this example, you add notification configuration to your Amazon S3 bucket to request Amazon S3 to publish events to AWS Lambda by invoking your Lambda function.

> **Important**
> Both the Lambda function and the Amazon S3 bucket must be in the same AWS region. This exercise assumes the `us-west-2` region.

Next Step

Step 1: Create a Lambda Function and Invoke it Manually Using the Console (Node.js) (p. 20)

Step 1: Create a Lambda Function and Invoke it Manually Using the Console (Node.js)

In this exercise, you will use the AWS Lambda console. You will use the **s3-get-object** blueprint and event data provided in the console.

- The blueprint provides a sample Amazon S3 object-created event to test your function. The event includes information such as bucket name, object key, and the size of the object as shown in the following.

```
{
  "Records": [
    {
        ...
      },
      "s3": {
         ...
        "bucket": {
          "name": "sourcebucket",
           ...
          "arn": "arn:aws:s3:::sourcebucket"
        },
        "object": {
          "key": "object keyname",
          "size": 1024,
          ...
        }
```

```
      }
    }
  ]
}
```

- The blueprint provides the following Node.js example code to process Amazon S3 events. You will use this sample code to create your Lambda function.

  In the code, the `handler` receives the event as the first parameter. The handler then processes the event, for illustration it simply writes the incoming event data to Amazon CloudWatch Logs, each `console.log()` statement generates log events in CloudWatch.

```
console.log('Loading function');
var aws = require('aws-sdk');
var s3 = new aws.S3({apiVersion: '2006-03-01'});

exports.handler = function(event, context) {
    console.log('Received event:', JSON.stringify(event, null, 2));
    // Get the object from the event and show its content type
    var bucket = event.Records[0].s3.bucket.name;
   var key = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/g,
 " ")) ;
    s3.getObject({Bucket: bucket, Key: key}, function(err, data) {
        if (err) {
            console.log("Error getting object " + key + " from bucket " +
bucket +
                ". Make sure they exist and your bucket is in the same region
 as this function.");
            context.fail ("Error getting file: " + err)
        } else {
            console.log('CONTENT TYPE:', data.ContentType);
            context.succeed();
        }
    });
};
```

Step 1.1: Create a Lambda Function

Follow the procedure in the Getting Started 1 exercise to create a Lambda function using the following values:

- In **Step 1: Select blueprint**, choose the **s3-get-object** blueprint.
- In **Step 2: Configure event sources**, click **Skip**.

  In this exercise you create a Lambda function first and then test it using sample event data, so you don't have to configure an Amazon S3 bucket at this time. You will do that in the next section.
- In **Step 3: Configure function**, specify the following values:
  - For the IAM **Role**, create a role by choosing the **S3 execution role** from the list.

    This role template has predefined access policy that grants sufficient permissions for AWS actions that your Lambda function needs. You create a new IAM role and assign a name (for example, **`lambda_s3_exec_role`**) as the execution role.

- In the **Advanced settings** section, increase the **Timeout (s)** to 5 seconds to avoid any timeout issues, and increase **Memory (MB)** to 512 MB. The amount of memory and timeout is configurable and depends on the size of objects you are creating in your bucket.

For instructions, see Step 1: Create a Lambda Function (p. 17).

Step 1.2: Invoke the Lambda Function Manually

Now you can manually invoke your Lambda function using a sample Amazon S3 event provided by the console.

1. Write down your Amazon S3 bucket name and object key name.

   Although this manual test Amazon S3 does not publish any events, you still need an existing bucket name and object key name to use in the sample event data provided by the console. When your Lambda function executes, it retrieves the object and then logs its content type.
2. Follow the instructions in the Getting Started 1 exercise to test the Lambda function. For instructions, see Step 2: Invoke the Lambda Function Manually (p. 19). Note the following:

   - On the **Input sample event** page, you need to select the **S3 Put** event from the **Sample event** list. This provides you with a sample Amazon S3 event to test your Lambda function.
   - Remember to update the sample event by providing your bucket name and object key name so that your Lambda function can download the object.

Next Step

Step 2: Configure Amazon S3 as the Event Source Using the Console (Node.js) (p. 22)

Step 2: Configure Amazon S3 as the Event Source Using the Console (Node.js)

In the preceding section you tested your Lambda function by using sample event data and manually invoking it. In this section, you configure your Lambda function to respond to object-created events published by Amazon S3. You will do this by adding an event source to your Lambda function using the Lambda console. The console does the following:

- In AWS Lambda: Grants Amazon S3 permission to invoke the Lambda function by adding a permission in the access policy associated with the Lambda function.
- In Amazon S3: Adds notification configuration on the specific Amazon S3 bucket requesting Amazon S3 to invoke Lambda function when an object is created.

  For more information about Amazon S3 notification feature, see Configuring Notifications for Amazon S3 Events in the Amazon Simple Storage Service Developer Guide.

Step 2.1: Add Amazon S3 as the Event Source

Follow the steps in this section to configure your Lambda function to respond to Amazon S3 object-created events. First, you create a bucket in Amazon S3.

1. In the Amazon S3 console, create a bucket.

   In the next step, you will add notification configuration to this bucket.

Important
The bucket must be in the same region where you created your Lambda function.

For instructions, see Create a Bucket in the *Amazon Simple Storage Service Getting Started Guide*.

2. In the AWS Lambda console, select the Lambda function.

   a. On the **Lambda> Functions > *`function-name`*** page, choose the **Event sources** tab.
   b. On the **Event Sources** tab, choose **Add event source**.
   c. On the **Add event source** page, select your bucket from the **Bucket** list, and choose **Object created** from the **Event type** list, and then choose **Submit**.

      This completes the setup.

Step 2.2: Test the Setup

To test the end-to-end setup, do the following:

1. Upload an object to the Amazon S3 bucket.

   Amazon S3 detects the object-created event, and publishes the event by invoking your Lambda function.

   Then, AWS Lambda executes your function by passing the Amazon S3 event to the Lambda function `handler`. The handler processes the event and writes some of the event information as logs to Amazon CloudWatch Logs.
2. You can verify your logs by using the following steps:

   a. Review function logs in the AWS Lambda console. To see specific function logs, choose the **Monitoring** tab.
   b. You can click the **logs** links to open and review the logs in the CloudWatch console.

Step 2.3: Cleanup

If you don't need the resources you created, you can remove them.

- Remove your Amazon S3 bucket using the Amazon S3 console.
- Remove the Lambda function using the AWS Lambda console.
- Remove IAM roles using the IAM console.

Related Topics

In this exercise, you used the AWS Lambda console for the setup. This guide also provides a follow-up walkthrough in which you perform all these steps using the AWS Command Line Interface (CLI). For more information, see AWS Lambda Walkthrough 2: Handling Amazon S3 Events Using the AWS CLI (Node.js) (p. 43).

Getting Started 3: Handling Amazon Kinesis Events Using the AWS Lambda Console (Node.js)

In this exercise, you create a Lambda function to consume events from an Amazon Kinesis stream.

Note
This is an example of the "pull" model where AWS Lambda polls the Amazon Kinesis stream and invokes your Lambda function when it detects new data on the stream. For more information, see AWS Lambda: How it Works (p. 3).

In this case, AWS Lambda both invokes and executes the Lambda function. The handler in your Lambda function gets the event data as parameter. It then processes the event, for illustration, logs some of the event information to Amazon CloudWatch.

- You will perform tasks in this exercise in two steps:
  - Create a Lambda function and manually invoke it using sample Amazon Kinesis event.
  - Add an event source. This is where you configure your Lambda function to respond to your Amazon Kinesis stream events.

    In a "pull" model, AWS Lambda manages the event source mappings. In this example, you will create event source mapping in AWS Lambda. Each of these event source mapping in AWS Lambda stores information such as an Amazon Kinesis stream ARN that is the event source and Lambda function to invoke when AWS Lambda detects events on the stream.

Important
Both the Lambda function and the Amazon Kinesis stream must be in the same AWS region. This exercise assumes the `us-west-2` region.

Next Step

Step 1: Create a Lambda Function and Invoke It Manually Using the Console (Node.js) (p. 24)

Step 1: Create a Lambda Function and Invoke It Manually Using the Console (Node.js)

In this exercise, you will use the AWS Lambda console. You will use the **kinesis-process-record** blueprint and event data provided in the console.

- The blueprint provides the following sample Amazon Kinesis event to test your function.

```
{
  "Records": [
    {
      "kinesis": {
        "partitionKey": "partitionKey-3",
        "kinesisSchemaVersion": "1.0",
        "data": "SGVsbG8sIHRoaXMgaXMgYSB0ZXN0IDEyMy4=",
        "sequenceNumber": "value"
      },
      "eventSource": "aws:kinesis",
      "eventID": "shardId-id",
      "invokeIdentityArn": "arn:aws:iam::account-id:role/testLEBRole",
      "eventVersion": "1.0",
      "eventName": "aws:kinesis:record",
      "eventSourceARN": "arn:aws:kinesis:us-east-1:account-id:stream/ex
amplestream",
      "awsRegion": "us-east-1"
    }
  ]
}
```

- The blueprint provides the following Node.js example code to process Amazon Kinesis events. You will use this sample code to create your Lambda function.

  In the code, the `handler` receives the event as the first parameter. The handler then processes the event, for illustration it simply writes payload (data on the stream) to Amazon CloudWatch Logs, each `console.log()` statement generates log events in CloudWatch.

```
console.log('Loading function');

exports.handler = function(event, context) {
    console.log('Received event:', JSON.stringify(event, null, 2));
    event.Records.forEach(function(record) {
        // Kinesis data is base64 encoded so decode here
       payload = new Buffer(record.kinesis.data, 'base64').toString('ascii');

        console.log('Decoded payload:', payload);
    });
    context.succeed("Successfully processed " + event.Records.length + " re
cords.");
};;
```

Step 1.1: Create a Lambda Function to Process Amazon Kinesis Stream Events

Follow the procedure in the Getting Started 1 exercise to create a Lambda function using the following values:

- In **Step 1: Select blueprint**, choose the **kinesis-process-record** blueprint.
- In **Step 2: Configure event sources**, click **Skip**.

  In this exercise you create a Lambda function first, and then test it using sample event data, so you don't have to configure an Amazon Kinesis stream as the event source at this time. You will do that in the next section.
- In **Step 3: Configure function**, specify the following values:
  - For the IAM **Role**, create a role by choosing the **Kinesis execution role** from the list.

    This role template has predefined access policy that grants sufficient permissions for AWS actions that your Lambda function needs. You create a new IAM role and assign a name (for example, **`lambda_kinesis_role`**) as the execution role.

For instructions, see Step 1: Create a Lambda Function (p. 17).

Step 1.2: Invoke the Lambda Function Manually

Now you can manually invoke your Lambda function using a sample Amazon Kinesis event provided by console. Follow the Getting Started 1 exercise for instructions to test the Lambda function. For instructions, see Step 2: Invoke the Lambda Function Manually (p. 19). Note the following:

- On the **Input sample event** page, you need to select the **Kinesis** event from the **Sample event** list. This provides you with a sample Amazon Kinesis event to test your Lambda function.

Next Step

Step 2: Configure an Amazon Kinesis Stream as the Event Source Using the Console (Node.js) (p. 26)

Step 2: Configure an Amazon Kinesis Stream as the Event Source Using the Console (Node.js)

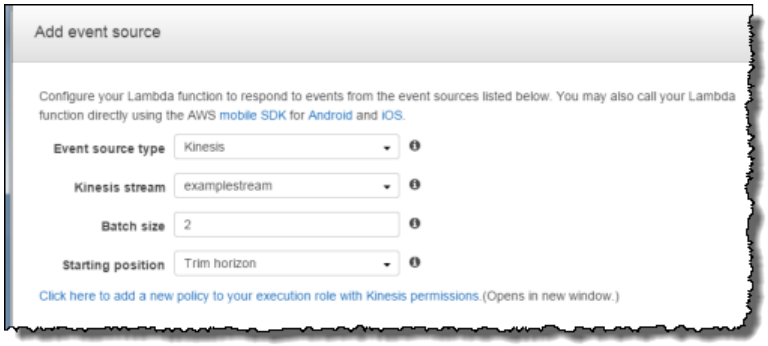
In the preceding section you tested your Lambda function by using a sample Amazon Kinesis event and manually invoking it. In this section, you configure your Lambda function to respond to Amazon Kinesis stream events. You will do this by adding an event source to your Lambda function using the Lambda console. The console creates an event source mapping in AWS Lambda that stores information including, the Amazon Kinesis stream that AWS Lambda needs to poll for events, the Lambda function to invoke when events are detected on the stream.

Step 2.1: Add an Amazon Kinesis Stream as the Event Source

First, you create an Amazon Kinesis stream using the Amazon Kinesis console. Then, you add the stream as the event source for your Lambda function using the AWS Lambda console.

1. Create a stream in the Amazon Kinesis console and set the number of shards to 1.
2. Add an event source in the AWS Lambda.

    a. In the AWS Lambda console, select the Lambda function.
    b. On the **Lambda> Functions >** ***`function-name`*** page, choose the **Event sources** tab.
    c. On the **Event Sources** tab, choose **Add event source**.
    d. On the **Add event source** page, choose **Kinesis** from the **Event source type** list, choose **stream** from the **Kinesis stream** list, set **Batch size** to **2**, and choose **Submit**.



This completes the setup.

Step 2.2: Test the Setup

To test the end-to-end experience, you need to upload sample records using the AWS CLI (you cannot use the console to add sample records to the stream). Using the following AWS CLI command, add event records to your Amazon Kinesis stream. The `--data` value is a base64-encoded value of the `"Hello, this is a test."` string .

```
aws kinesis put-record \
--stream-name examplestream \
--data "Hello, this is a test" \
--partition-key shardId-000000000000 \
--region aws-region, for example us-west-2 \
--profile adminuser
```

For example walkthroughs that use AWS CLI for setting up and testing, see Step 1: Prepare for the Walkthrough (p. 36).

You can run the same command more than once to add multiple records to the stream. The following things happen:

- AWS Lambda polls the stream. When it detects updates on the stream, it invokes your Lambda function.
- AWS Lambda then executes your Lambda function by passing the event to the `handler` in your Lambda function.

  The handler processes the event and writes some of the event information as logs to Amazon CloudWatch Logs.

You can review the logs in the AWS Lambda console. To see specific function logs, choose **Monitoring** tab. You can click the **logs** links to open and review the logs in the CloudWatch console.

Step 2.3: Cleanup

If you don't need the resources you created, you can remove them.

- Remove the Amazon Kinesis stream.
- Remove the Lambda function using the AWS Lambda console.
- Remove the IAM role using the IAM console.

Related Topics

In this exercise, you completed the setup using the console. This guide also provides a follow-up walkthrough in which you perform all these steps using the AWS Command Line Interface (CLI). For more information, see AWS Lambda Walkthrough 4: Processing Events from an Amazon Kinesis Stream Using the AWS CLI (Node.js) (p. 60).

Getting Started 4: Creating an HTTP Endpoint-Enabled Lambda Function (AWS Lambda Integration with Amazon API Gateway)

This exercise shows you how to create an HTTP endpoint-enabled Lambda function.

Scenario

In this exercise, you will create a Lambda function using the following Node.js code example. The Lambda function supports various DynamoDB table operations such as create an item, update an item, and delete an item. For easy testing, it also supports the `echo` operation, which simply echos the incoming event.

```
console.log('Loading function');

var doc = require('dynamodb-doc');
var dynamo = new doc.DynamoDB();

exports.handler = function(event, context) {
    //console.log('Received event:', JSON.stringify(event, null, 2));

    var operation = event.operation;
```

```
    delete event.operation;

    switch (operation) {
        case 'create':
            dynamo.putItem(event, context.done);
            break;
        case 'read':
            dynamo.getItem(event, context.done);
            break;
        case 'update':
            dynamo.updateItem(event, context.done);
            break;
        case 'delete':
            dynamo.deleteItem(event, context.done);
            break;
        case 'list':
            dynamo.scan(event, context.done);
            break;
        case 'echo':
            context.succeed(event);
            break;
        case 'ping':
            context.succeed('pong');
            break;
        default:
            context.fail(new Error('Unrecognized operation "' + operation +
'"'));
    }
};
```

Next, you will configure an HTTP endpoint for the Lambda function using Amazon API Gateway.

Lastly, you will test it by sending an HTTP request to the Lambda function endpoint. You will do this using the Amazon API Gateway console. You will send a request body with your POST request. Amazon API Gateway will invoke your Lambda function and AWS Lambda will execute the function on your behalf. The HTTP POST request body you send is provided to your Lambda function `handler` as event data. Then, the `handler` will process the event.

Implementation Summary

You will create, configure, and test the setup using the AWS Lambda console and the Amazon API Gateway console as follows:

1. In the AWS Lambda console, you will use the blueprint called *microservice-http-endpoint* to configure the Lambda function and configure the HTTP endpoint for that function.

   To configure an HTTP endpoint, the blueprint creates an API using Amazon API Gateway, and then adds the Lambda function as a resource so that you can send an HTTP POST request to the Lambda function endpoint.

   Note that the Lambda console also configures the required permissions as follows:

   - Grants the Amazon API Gateway service permission to invoke your Lambda function by adding a permission in the access policy associated with your Lambda function.
   - Creates an IAM role (called an *execution role*) that grants AWS Lambda permission to assume the role, so that the AWS Lambda service can execute your Lambda function on your behalf. The access policy associated with the execution role grants permissions for the DynamoDB actions that your Lambda function will need to perform.

To learn more about invocation and execution permissions, see Permission Model (p. 8).

2. In the Amazon API Gateway console, you will test the setup by sending an HTTP POST request to the Lambda function endpoint with a sample request body. Amazon API Gateway invokes your Lambda function, and then AWS Lambda executes your Lambda function on your behalf by passing the request body as an event parameter to your Lambda function `handler`. Then, the `handler` processes the event.

   An easy way to test the endpoint is to send an HTTP POST request with `echo` as the operation, which causes the Lambda function to simply echo the incoming event (the results appear in the console). In addition, the Lambda function also supports DynamoDB table operations, such as create an item, delete an item, and update an item. You need to make sure you create the table first before you try to perform these operations.

   Important
   This exercise assume you are creating AWS resources, your Lambda function, API in the **us-east-1** AWS region.

Next Step

Step 1: Create a Lambda Function and Invoke It Manually Using the Console (Node.js) (p. 29)

Step 1: Create a Lambda Function and Invoke It Manually Using the Console (Node.js)

You will perform this exercise using the AWS Lambda console. You will use the blueprint called *microservice-http-endpoint* and event data provided in the AWS Lambda console.

Step 1.1: Create a Lambda Function

Follow the procedure in Getting Started 1: Invoking Lambda Functions from User Applications Using the AWS Lambda Console (Node.js) (p. 16) to create a Lambda function using the following values:

- In **Step 1: Select blueprint**, choose the **microservice-http-endpoint** blueprint.
- In **Step 2: Configure function**, specify the following values:
  - Enter a function name (for example, **`ExampleFunctionWithHTTPEndpoint`**).
  - Select Node.js from the **Runtime** list.
  - For the IAM **Role**, create a role by choosing the **Basic with DynamoDB** from the list.

    This role template has a predefined access policy that grants sufficient permissions for AWS actions that your Lambda function needs. Create a new IAM role and assign a name (for example, **`lambda_dynamo`**) as the execution role.
- In **Step 3: Configure endpoints**, specify the following values:
  - Leave the default **API name** as **LambdaMicroservice**. After the API is created, it will appear in the Amazon Gateway console.
  - Leave the **Resource name** as it appears in the console. It maps to the name that you specified for your Lambda function, so do not change this value.
  - Choose **POST** from the **Method** list.
  - Leave the default **prod** for **Deployment stage**.
  - Choose **Open with access key** from the **Security** list.

Step 1.2: Invoke the Lambda Function Manually

You can manually invoke the Lambda function in the AWS Lambda console. You will use the following JSON as a sample event:

```
{
   "operation": "echo",
   "somekey1": "somevalue1",
   "somekey2": "somevalue2"
}
```

Follow the procedure in the Getting Started 1 exercise to test the Lambda function. For instructions, see Step 2: Invoke the Lambda Function Manually (p. 19). Note the following:

- On the **Input sample event** page, choose the **Hello World** event from the **Sample event** list, and replace the sample event with the JSON listed in this step.

Next Step

Step 2: Invoke the Lambda Function Using the HTTP Endpoint (p. 30)

Step 2: Invoke the Lambda Function Using the HTTP Endpoint

In this section you send an HTTP POST request to your Lambda function endpoint.

1. In this exercise, you will use the following JSON in the request body:

   ```
   {
      "operation": "echo",
      "somekey1": "somevalue1",
      "somekey2": "somevalue2"
   }
   ```

   When the Lambda function executes, the handler gets this JSON as the event parameter and then it processes the event. For the `echo` operation, the Lambda function simply echos the incoming event.

   a. In the AWS Lambda console, select the Lambda function.
   b. In the **API endpoints** tab, click the POST **method** link.

      This opens the Amazon API Gateway console, showing the POST method page for your Lambda function endpoint.
   c. Choose **Test**.
   d. Open the **Request Body** section and provide the following JSON:

      ```
      {
         "operation": "echo",
         "somekey1": "somevalue1",
         "somekey2": "somevalue2"
      }
      ```

   e. Choose **Test**.

Amazon API Gateway invokes your Lambda function. AWS Lambda executes your Lambda function on your behalf. The Lambda function handler processes the incoming event, which in this case simply echos the incoming event. You can also verify that the response is same as what you saw earlier in the Lambda console.

2. (Optional) In this case, you send JSON payload with request for a `create` operation to create an item in a DynamoDB table. This will require you to first create a table in DynamoDB and then invoke the Lambda function.

   a. In the DynamoDB console (https://console.aws.amazon.com/dynamodb) create a table (`myTable`) with a primary key made of only a hash attribute (`id`) of string type.

   b. Now, send an HTTPS POST request to the Lambda function endpoint using following JSON in the request body:

```
{
    "operation": "create",
    "TableName": "myTable",
    "Item": {
        "id": "test"
    }
}
```

   c. After you send the request, the Lambda function executes and creates an item in the table. You can verify the new item in the DynamoDB console.

Creating Deployment Package (Node.js)

To create a Lambda function you first create a Lambda function deployment package, a .zip file consisting of your code and any dependencies. At the time you create a Lambda function you provide the .zip file and other configuration information such as name, description, and run-time requirements like memory allocation. If you want, you can upload the .zip file first to an Amazon S3 bucket in the same AWS region where you want to create the Lambda function, and then specify the bucket name and object key name when you create the Lambda function.

Depending on the resources your custom code uses, you have the following options when creating a Lambda function:

- Simple scenario—If your custom code requires only the AWS SDK library, then you can use the inline editor in the AWS Lambda console. Using the console, you can edit and upload your code to AWS Lambda. The console will zip up your code with the relevant configuration information into a deployment package that the Lambda service can run.

  You can also test your code in the console by manually invoking it using sample event data.

  Note
  The Lambda service has preinstalled the AWS SDK for Node.js.

- Advanced scenario—If you are writing code that uses other resources, such as a graphics library for image processing, or you want to use the AWS CLI instead of the console, you need to first create the Lambda function deployment package, and then use the console or the CLI to upload the package.

Programming Model (Node.js)

Your Lambda function code must be written in a stateless style, and have no affinity with the underlying compute infrastructure. Your code should expect local file system access, child processes, and similar artifacts to be limited to the lifetime of the request, and store any persistent state in Amazon S3, Amazon DynamoDB, or another cloud storage service. Requiring functions to be stateless enables AWS Lambda to launch as many copies of a function as needed to scale to the incoming rate of events and requests. These functions may not always run on the same compute instance from request to request, and a given instance of your Lambda function may be used more than once by AWS Lambda.

The following example skeleton code shows the format in which you write your custom Node.js code:

```
exports.handler_name = function(event, context) {
   console.log("value1 = " + event.key1);
   console.log("value2 = " + event.key2);
   ...
  context.succeed("some message");
}
```

In the code:

- *handler_name* – The `handler` parameter name you provided when you create a Lambda function (see `CreateFunction`). This tells Lambda which Node.js function to call.
- The `event` parameter – Lambda passes event data to the function via this parameter. The structure of event data depends on the event source. For example, if Amazon S3 is the event source, the event data will provide, among other things, bucket name and object key. For more information about Amazon S3's event data structure, go to Event Message Structure in the *Amazon Simple Storage Service Developer Guide*.
- The `context` parameter – See the following section.

The `context` Object: Methods and Properties

You interact with AWS Lambda execution environment via the `context` parameter. The `context` object allows you to specify when the function and any callbacks have completed execution. It also allows you to access useful information available within the Lambda execution environment. For example, you can use the `context` parameter to determine the CloudWatch log stream associated with the function, or use the `clientContext` property of the `context` object to learn more about the application calling the Lambda function (when invoked through the AWS Mobile SDK).

You can use the `Context` object methods as follows:

`context.succeed()` method

Indicates the Lambda function execution and all callbacks completed successfully. Here's the general syntax:

```
context.succeed (Object result);
```

where

`Object result` – provides the result of the function execution. The `result` provided must be `JSON.stringify` compatible. This parameter is optional. You can call this method without any parameters

(succeed()) or pass a null value (succeed(null)). If AWS Lambda fails to stringify or encounters another error, an unhandled error is thrown, with the `X-Amz-Function-Error` response header set to `Unhandled`.

The method behavior depends on the invocation type specified when the Lambda function is invoked (see Invoke (p. 178)).

- If the Lambda function is invoked using the `Event` invocation type, this `succeed` method returns HTTP status 202, request accepted.
- If the Lambda function is invoked using the `RequestResponse` invocation type, this `succeed` method will return HTTP status 200 (OK) and set the response body to the string representation of result.

`context.fail()` method

Indicates the Lambda function execution and all callbacks completed unsuccessfully, resulting in a handled exception. Here's the general syntax:

```
context.fail (Error error);
```

where

`Error error` – provides the result of the Lambda function execution.

The parameter is optional. You can call this method without the parameter or pass null as a parameter value. Non-null error values will populate the response body.

The `fail` method will set the response body to the string representation of error and also write to logs.

If AWS Lambda fails to stringify or encounters another error, an unhandled error, with the `X-Amz-Function-Error` header set to `Unhandled`.

> **Note**
> For the error from the `context.done(error, null)` and `context.fail(error)`, Lambda will log the first 256 KB of the error object. In case of a larger error object, it will be truncated and logged, and the customers should see the text `- Truncated by Lambda` next to their error object.

`context.done()` method

Causes the Lambda function execution to terminate. This method complements the `context.succeed()` and `context.fail()` methods by allowing the use of the "error first" callback design pattern. It provides no additional functionality.

Here's the general syntax:

```
context.done (Error error, Object result);
```

where

`Error error` – provides an error representing the results of the failed Lambda function execution.

`Object result` – provides the result of a successful function execution. The result provided must be `JSON.stringify` compatible. If an error is provided, this parameter is expected to be null.

Both the parameters are optional. You can call this method without any parameters or pass null as a parameter value.

AWS Lambda treats any non-null value for the `error` parameter as a managed exception.

The function behavior depends on the invocation type specified when the Lambda function is invoked (see Invoke (p. 178)).

- If the Lambda function was invoked using the `Event` or the `RequestResponse` invocation type, this `done` method automatically logs the string representation of non-null values of `error` to the Amazon CloudWatch Logs stream associated with the Lambda function.
- If the Lambda function was invoked using the `RequestResponse` invocation type, the `done` method will do the following:
  - If the error is null, set the response body to the string representation of `result`, similar to the `context.fail()`.
  - If the error is not null, set the response body to `error`.
  - If the function is called with a single argument of type error, the error value will be populated in the response body.

Note
For the error from the `context.done(error, null)` and `context.fail(error)`, Lambda will log the first 256 KB of the error object, and in case of a larger error object, it will be truncated and logged and the customers should see the text `- Truncated by Lambda` next to their error object.

`context.getRemainingTimeInMillis` method

Returns the approximate remaining execution time (before timeout occurs) of the Lambda function that is currently executing. At the time you create your Lambda function you set the timeout and when the timeout reaches AWS Lambda terminates your Lambda function. You can use this method to check the remaining time during your function execution and take appropriate corrective action.

Here's the general syntax:

```
context.getRemainingTimeInMillis ();
```

`context` Object Properties

The `context` object properties are:

- awsRequestId – The request ID for the Lambda function invocation request that is currently being executed.
- logStreamName – The CloudWatch log stream name associated with the invoked Lambda function.
- clientContext – Information about the client application and device when invoked through the AWS Mobile SDK. It can be null.
- identity – Information about the Amazon Cognito identity provider when invoked through the AWS Mobile SDK. It can be null.
- logGroupName–The name of the CloudWatch log group where you can find log output for the function.
- logStreamName–The name of the CloudWatch log stream where you can find log output for the function execution. The log stream may or may not change for each invocation of the Lambda function.
- functionName–The name of your Lambda function that is being executed.

Related Topics

AWS Lambda: How it Works (p. 3)

AWS Lambda Walkthroughs (Node.js)

This section contains several AWS CLI–based examples that walk through how to use AWS Lambda. The walkthroughs show you how to create Lambda functions and invoke them in response to Amazon S3 event notifications, Amazon DynamoDB table updates, and Amazon Kinesis streams. They also illustrate how to use Lambda functions with a user application or to develop an Alexa Skill for Amazon Echo.

Topics

In addition, for an example walkthrough of using AWS Lambda functions with Amazon Echo, go to Developing an Alexa Skill as a Lambda Function.

AWS Lambda Walkthrough 1: Invoking Lambda Functions from User Applications Using the AWS CLI (Node.js)

One of the use cases for using AWS Lambda is to process events generated by a user application. For demonstration purposes, you don't need to write a user application that will invoke your Lambda function. Instead, in this walkthrough, you will use sample event data and invoke your Lambda function manually.

When a user application invokes a Lambda function as shown in this walkthrough, it's an example of the AWS Lambda *request-response* model in which an application invokes a Lambda function and receives a response in real time. For more information, see AWS Lambda: How it Works (p. 3).

> **Note**
> In this walkthrough, you will use the AWS CLI to create and invoke a Lambda function and explore other AWS Lambda APIs.

Here's an overview of what you'll be doing:

- Create a Lambda function to process an event it receives as a parameter. You will use the following example JavaScript code to create your Lambda function.

```
console.log('Loading function');

exports.handler = function(event, context) {
    console.log('value1 =', event.key1);
    console.log('value2 =', event.key2);
    console.log('value3 =', event.key3);
    context.succeed(event.key1);  // Echo back the first key value
    // context.fail('Something went wrong');
};
```

The function is simple. It processes incoming event data by logging it; these logs are available in Amazon CloudWatch, and in the request-response model, you can request the log data be returned in the response.

- Simulate a user application sending an event to your Lambda function by invoking your Lambda function manually using the following sample event data.

```
{
  "key1": "value1",
  "key2": "value2",
  "key3": "value3"
}
```

Note
This example is similar to the getting started example (see Getting Started 1: Invoking Lambda Functions from User Applications Using the AWS Lambda Console (Node.js) (p. 16)). The difference is that the Getting Started exercises provide a console-based experience. The console does many things for you, simplifying your experience. When using the CLI, you get the raw experience of making the API calls, which can help you familiarize yourself with the AWS Lambda operations. In addition to creating and invoking a Lambda function, you will explore other Lambda APIs.

Next Step

Step 1: Prepare for the Walkthrough (p. 36)

Step 1: Prepare for the Walkthrough

You need to set up the AWS CLI to test this walkthrough. The exercise assumes that you are using administrator user credentials. We refer to the administrator user as *adminuser* in this walkthrough. For instructions on creating an administrator user in your AWS account, see Set Up an AWS Account and Create an Administrator User (p. 12).

Set Up the AWS CLI

Before you can start the example walkthrough, you need to download and configure the AWS Command Line Interface (CLI).

To set up the AWS CLI

1. Download and configure the AWS CLI. For instructions, see the following topics in the *AWS Command Line Interface User Guide*.

   - Getting Set Up with the AWS Command Line Interface
   - Configuring the AWS Command Line Interface

2. Add a named profile for the administrator user in the CLI config file. You will use this profile when executing the CLI commands.

```
[profile adminuser]
aws_access_key_id = adminuser access key ID
aws_secret_access_key = adminuser secret access key
region = aws-region
```

For a list of available AWS regions, go to Regions and Endpoints in the *AWS General Reference*.

3. Verify the setup by entering the following commands at the command prompt.

   - Try the help command to verify that the AWS CLI is installed on your computer:

     ```
     aws help
     ```

   - Try a Lambda command to verify the user can reach AWS Lambda. This command lists Lambda functions in the account, if any. The AWS CLI uses the `adminuser` credentials to authenticate the request.

     ```
     aws lambda list-functions --profile adminuser
     ```

Next Step

Step 2: Create a Lambda Function (p. 37)

Step 2: Create a Lambda Function

In this section, you first do the following:

- Create a deployment package — A deployment package is a .zip file containing your code and any dependencies. For this walkthrough there are no dependencies, you only have a simple example code.
- Create an IAM role (execution role) — At the time you upload you deployment package to create your Lambda function, you must specify an IAM role. AWS Lambda uses this role when executing your function.

  You also grant this role the permissions your Lambda function needs. The code in this walkthrough writes logs to Amazon CloudWatch Logs. So you grant permission for CloudWatch actions. For more information, see Amazon LambdaWatch Logs.

You will then create a Lambda function ("HelloWorld") using the `create-function` CLI command. For more information about the underlying API and related parameters, see CreateFunction (p. 160).

Step 2.1: Create a Lambda Function Deployment Package

Follow the instructions to create an AWS Lambda function deployment package.

1. Open a text editor, and copy the following code.

   ```
   console.log('Loading function');

   exports.handler = function(event, context) {
       console.log('value1 =', event.key1);
       console.log('value2 =', event.key2);
       console.log('value3 =', event.key3);
       context.succeed(event.key1);  // Echo back the first key value
       // context.fail('Something went wrong');
   };
   ```

2. Save the file as `helloworld.js`.

3. Zip the `helloworld.js` file as `helloworld.zip`.

Step 2.2: Create an IAM Role (execution role)

When the Lambda function in this exercise executes, it will need permissions to write logs to Amazon CloudWatch. You grant these permission by creating an IAM role (also referred as an execution role). AWS Lambda assumes this role when executing your Lambda function on your behalf. In this section, you create an IAM role using the following predefined role type and access policy:

- AWS service role of the "AWS Lambda" type. This role grants AWS Lambda permission to assume the role.
- "AWSLambdaBasicExecutionRole" access policy that you attach to the role. This existing policy grants permissions that include permissions for Amazon CloudWatch actions that your Lambda function needs.

For more information about IAM roles, go to Creating a Role for an AWS Service in *Using IAM*.

To create an IAM role (*executionrole*)

1. Sign in to the AWS Management Console.
2. In the IAM console, create an IAM role, `executionrole`. As you follow the steps to create a role, note the following:

   - In **Select Role Type**, click **AWS Service Roles**, and then select **AWS Lambda**.
   - In **Attach Policy**, select the policy named **AWSLambdaBasicExecutionRole**.

   For instructions, go to Creating a Role for an AWS Service (AWS Management Console) in *IAM User Guide*.
3. Write down the Amazon Resource Name (ARN) of the IAM role. You will need this value when you create your Lambda function in the next step.

Step 2.3: Create a Lambda Function

Execute the following Lambda CLI `create-function` command to create a Lambda function. You provide the deployment package and IAM role ARN as parameters.

```
$ aws lambda create-function \
--region us-west-2 \
--function-name helloworld \
--zip-file fileb://file-path/helloworld.zip \
--role role-arn \
--handler helloworld.handler \
--runtime nodejs \
--profile adminuser
```

Note that if you want you can upload the .zip file to an Amazon S3 bucket in the same AWS region, and then specify the bucket and object name in the preceding command. You will need to replace the `--zip-file` parameter by the `--code` parameter as shown:

```
--code S3Bucket=bucket-name,S3Key=zip-file-object-key
```

For more information, see CreateFunction (p. 160). AWS Lambda creates the function and returns function configuration information as shown in the following example:

```
{
    "FunctionName": "helloworld",
    "CodeSize": 351,
    "MemorySize": 128,
    "FunctionArn": "function-arn",
    "Handler": "helloworld.handler",
    "Role": "arn:aws:iam::account-id:role/LambdaExecRole",
    "Timeout": 3,
    "LastModified": "2015-04-07T22:02:58.854+0000",
    "Runtime": "nodejs",
    "Description": ""
}
```

Next Step

Step 3: Invoke the Lambda Function (p. 39)

Step 3: Invoke the Lambda Function

In this section, you invoke your Lambda function manually using the invoke CLI command.

```
$ aws lambda invoke \
--invocation-type RequestResponse \
--function-name helloworld \
--region us-west-2 \
--log-type Tail \
--payload '{"key1":"value1", "key2":"value2", "key3":"value3"}' \
--profile adminuser \
outputfile.txt
```

If you want you can save the payload to a file (say "input.txt") and provide the file name as a parameter.

```
--payload file://input.txt \
```

The preceding `invoke` command specifies "RequestResponse" as the invocation type, which returns a response immediately in response to the execution. You can alternatively specify "Event" as the invocation type to invoke the function asynchronously.

By specifying the `--log-type` parameter, the command also requests the tail end of the log produced by the function. The log data in the response is base64-encoded as shown in the following example response:

```
{
     "LogResult": "base64-encoded-log",
     "StatusCode": 200
}
```

On Linux and Mac, you can use the base64 command to decode the log.

```
$ echo base64-encoded-log | base64 --decode
```

The following is a decoded version of an example log.

```
START RequestId: 16d25499-d89f-11e4-9e64-5d70fce44801
2015-04-01T18:44:12.323Z    16d25499-d89f-11e4-9e64-5d70fce44801    value1 =
value1
2015-04-01T18:44:12.323Z    16d25499-d89f-11e4-9e64-5d70fce44801    value2 =
value2
2015-04-01T18:44:12.323Z    16d25499-d89f-11e4-9e64-5d70fce44801    value3 =
value3
2015-04-01T18:44:12.323Z    16d25499-d89f-11e4-9e64-5d70fce44801    result:
"value1"
END RequestId: 16d25499-d89f-11e4-9e64-5d70fce44801
REPORT RequestId: 16d25499-d89f-11e4-9e64-5d70fce44801
Duration: 13.35 ms      Billed Duration: 100 ms   Memory Size: 128 MB
Max Memory Used: 9 MB
```

For more information, see Invoke (p. 178).

Because you invoked the function using the "RequestResponse" invocation type, the function executes and returns the object you passed to the `context.succeed()` in real time when it is called. In this example, you will see the following text written to the `outputfile.txt` you specified in the CLI command:

```
"value1"
```

Note
You are able to execute this function because you are using the same AWS account to create and invoke the Lambda function. However, if you want to grant cross-account permission to another AWS account or an AWS service permission to execute the function, you must add a permission to the access policy associated with the function. The walkthrough that uses Amazon S3 as the event source (see AWS Lambda Walkthrough 2: Handling Amazon S3 Events Using the AWS CLI (Node.js) (p. 43)) grants such permission to Amazon S3 to invoke the function.

You can monitor the activity of your Lambda function in the AWS Lambda console.

- The AWS Lambda console shows a graphical representation of some of the CloudWatch metrics in the **Cloudwatch Metrics at a glance** section for your function. Sign in to the AWS Management Console at https://console.aws.amazon.com/.
- For each graph, you can also click the **logs** link to view the CloudWatch logs directly.

Next Step

Step 4: Try More CLI Commands (p. 40)

Step 4: Try More CLI Commands

Step 4.1: List the Lambda Functions in Your Account

In this section, you try AWS Lambda list function operations. Execute the following CLI `list-functions` command to retrieve a list of functions you uploaded.

```
$ aws lambda list-functions \
--max-items 10 \
--profile adminuser
```

To illustrate the use of pagination, the command specifies the optional `--max-items` parameter to limit the number of functions returned in the response. For more information, see ListFunctions (p. 185). The following is an example response.

```
{
    "Functions": [
        {
            "FunctionName": "helloworld",
            "MemorySize": 128,
            "CodeSize": 412,
            "FunctionArn": "arn:aws:lambda:us-east-1:account-id:function:Pro
cessKinesisRecords",
            "Handler": "ProcessKinesisRecords.handler",
            "Role": "arn:aws:iam::account-id:role/LambdaExecRole",
            "Timeout": 3,
            "LastModified": "2015-02-22T21:03:01.172+0000",
            "Runtime": "nodejs",
            "Description": ""
        },
        {
            "FunctionName": "ProcessKinesisRecords",
            "MemorySize": 128,
            "CodeSize": 412,
            "FunctionArn": "arn:aws:lambda:us-east-1:account-id:function:Pro
cessKinesisRecords",
            "Handler": "ProcessKinesisRecords.handler",
           "Role": "arn:aws:iam::account-id:role/lambda-execute-test-kinesis",

            "Timeout": 3,
            "LastModified": "2015-02-22T21:03:01.172+0000",
            "Runtime": "nodejs",
            "Description": ""
        },
        ...
      ],
       "NextMarker": null

}
```

In response, Lambda returns a list of up to 10 functions. If there are more functions you can retrieve, `NextMarker` provides a marker you can use in the next `list-functions` request; otherwise, the value is null. The following `list-functions` CLI command is an example that shows the `--next-marker` parameter.

```
$ aws lambda list-functions \
--max-items 10 \
--marker value-of-NextMarker-from-previous-response \
--profile adminuser
```

Step 4.2: Get Metadata and URL to Download Previously Uploaded Lambda Function Deployment Package

The Lambda CLI `get-function` command returns Lambda function metadata and a presigned URL that you can use to download the function's .zip file (deployment package) that you uploaded to create the function. For more information, see GetFunction (p. 171).

```
$ aws lambda get-function \
--function-name helloworld \
--region us-west-2 \
--profile adminuser
```

The following is an example response.

```
{
    "Code": {
        "RepositoryType": "S3",
        "Location": "pre-signed-url"
    },
   "Configuration": {
        "FunctionName": "helloworld",
        "MemorySize": 128,
        "CodeSize": 287,
       "FunctionArn": "arn:aws:lambda:us-west-2:account-id:function:helloworld",

        "Handler": "helloworld.handler",
        "Role": "arn:aws:iam::account-id:role/LambdaExecRole",
        "Timeout": 3,
        "LastModified": "2015-04-07T22:02:58.854+0000",
        "Runtime": "nodejs",
        "Description": ""
    }

}
```

If you want only the function configuration information (not the presigned URL), you can use the Lambda CLI `get-function-configuration` command.

```
$ aws lambda get-function-configuration \
 --function-name helloworld \
 --region us-west-2 \
--profile adminuser
```

Next Step

Step 5: Delete the Lambda Function and IAM Role (p. 42)

Step 5: Delete the Lambda Function and IAM Role

Execute the following `delete-function` command to delete `helloworld` function.

```
$ aws lambda delete-function \
 --function-name helloworld \
 --region us-west-2 \
--profile adminuser
```

Delete IAM Role

After you delete the Lambda function you can also delete the IAM role you created in the IAM console. For information about deleting a role, see Deleting Roles or Instance Profiles in *IAM User Guide*.

AWS Lambda Walkthrough 2: Handling Amazon S3 Events Using the AWS CLI (Node.js)

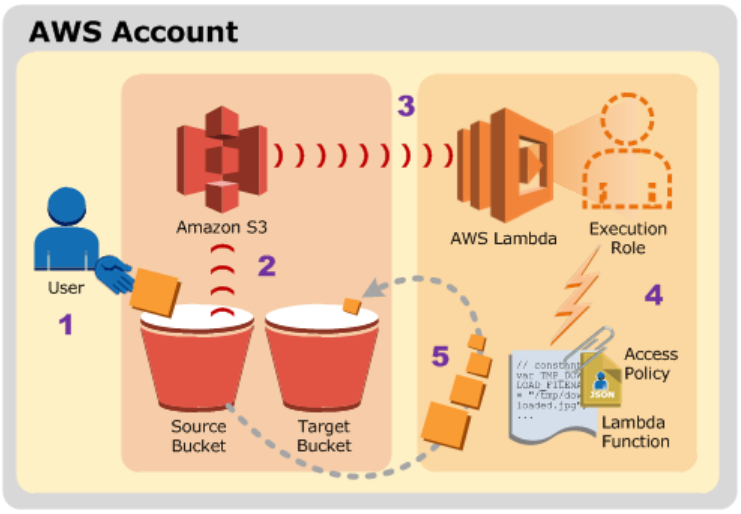
Scenario

Suppose you have two buckets in Amazon S3. You store images (.jpg and .png objects) in one bucket (*`sourcebucket`*), and for each object created in the bucket, you want AWS Lambda to execute a Lambda function to create a thumbnail in the *`sourcebucket`*`resized` bucket. You will use Amazon S3's bucket notification configuration feature to request Amazon S3 to publish object-created events to AWS Lambda. In the notification configuration, you will identify your Lambda function (called `CreateThumbnail`) that you want Amazon S3 to invoke.

Important
You must use two buckets. If you use the same bucket as source and target, each thumbnail uploaded to the source bucket will trigger another object-created event, which will invoke the Lambda function creating the unwanted recursion.

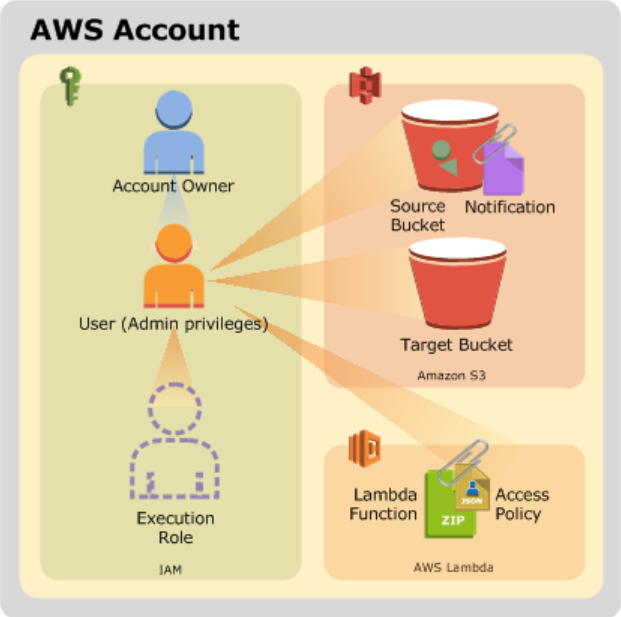
Implementation Summary

The following diagram illustrates the application flow:



1. A user uploads an object to the source bucket.
2. Amazon S3 detects the object-created event.
3. Amazon S3 publishes the `s3:ObjectCreated:*` event to AWS Lambda by invoking the Lambda function by passing event data as function parameter.
4. Lambda executes the function by assuming the execution role.
5. From the event data it received, the Lambda function knows the source bucket name and object key name. The Lambda function reads the object and creates a thumbnail using graphics libraries, and saves it to the target bucket.

Note that upon completing this exercise, you will have the following S3, Lambda, and IAM resources in your account.



As the diagram shows, you create Amazon S3, AWS Lambda, and AWS Identity and Access Management (IAM) resources in your account:

In Lambda:

- A Lambda function – Amazon S3 invokes this function by passing event data such as the source bucket name and object key as parameter. The function reads the object and, if the object is a .jpg or .png object, creates a thumbnail in the target bucket.
- An access policy – In the access policy associated with the Lambda function, you will add a permission granting Amazon S3 permission to invoke the Lambda function. You will also restrict the permission so that Amazon S3 can invoke the Lambda function only for object-created events from a specific bucket owned by a specific AWS account.

  Note
  It is possible for an AWS account to delete a bucket and some other AWS account to later create a bucket with same name. The additional conditions ensure that Amazon S3 can invoke the Lambda function only if Amazon S3 detects object-created events from a specific bucket owned by a specific AWS account.

  For more information, see AWS Lambda: How it Works (p. 3).

In IAM:

- Administrator user – Called *adminuser*. Using root credentials of an AWS account is not recommended. Instead, use the *adminuser* credentials to perform steps in this exercise.
- An IAM role (execution role) – When you create this role, you will trust AWS Lambda to assume this role. You will also grant sufficient permissions that your Lambda function needs—for example, permission to read objects from the source bucket and create a thumbnail in the target bucket.

In Amazon S3:

- Two buckets – We refer to these as the source bucket and the target bucket.
- Notification configuration – You will add notification configuration on your source bucket identifying the type of events (object-created events) you want Amazon S3 to publish to AWS Lambda and the Lambda

function to invoke. For more information about the Amazon S3 notification feature, go to Setting Up Notification of Bucket Events.

Now you are ready to try the steps. Note that after the initial preparation the walkthrough is divided into two main sections:

- First, you do the necessary setup to create a Lambda function and invoke it manually using Amazon S3 sample event data. This intermediate testing verifies that the function works.
- Then you will add a notification configuration to your source bucket so that Amazon S3 can then invoke your Lambda function when it detects object-created events.

Next Step

Step 1: Prepare for the Walkthrough (Amazon S3 Events) (p. 45)

Step 1: Prepare for the Walkthrough (Amazon S3 Events)

In this section, you do the following:

- Create an administrator user, *adminuser*. For instructions, see Set Up an AWS Account and Create an Administrator User (p. 12). If you have an administrator user, you can skip this step.
- Create two buckets with a sample .jpg object (`HappyFace.jpg`) in the source bucket.
- Set up the AWS CLI.

Step 1.1: Create Buckets and Upload a Sample Object

Follow the steps to create buckets and upload an object.

> **Important**
> Both the source bucket and your Lambda function must be in the same AWS region. In addition, the example code used for the Lambda function also assumes both the buckets are in the same region. This exercise assumes the `us-west-2` region.

1. Using the IAM User Sign-In URL, sign in to the Amazon S3 console as *adminuser*.
2. Create two buckets. The target bucket name must be *`sourcebucket`* followed by "resized". For example, "mybucket" and "mybucketresized".

   For instructions, go to Create a Bucket in the *Amazon Simple Storage Service Getting Started Guide*.
3. In the source bucket, upload a .jpg object, `HappyFace.jpg`.

   When you invoke the Lambda function manually, before hooking up Amazon S3, you will pass sample event data to the function that specifies the source bucket and `HappyFace.jpg` as the newly created object. So you need to create this sample object.

Step 1.2: Set Up the AWS CLI

You use the AWS CLI to upload your Lambda function deployment package. You will test the resulting Lambda function by manually invoking it using the CLI. Walkthrough 1 provides instructions to set up the AWS CLI and add the *adminuser* profile. For instructions, see Step 1: Prepare for the Walkthrough (p. 36).

Next Step

Step 2: Create and Test the Lambda Function (Amazon S3 Events) (p. 46)

Step 2: Create and Test the Lambda Function (Amazon S3 Events)

In this section, you do the following:

- Create a Lambda function deployment package using the sample Node.js code provided.
- Create an IAM role (execution role) — At the time you upload the deployment package, you will need to specify an IAM role (execution role) that Lambda can assume to execute the function on your behalf.
- Create the Lambda function and test.

Follow the instructions in the following sections:

Step 2.1: Create a Lambda Function Deployment Package (p. 46)

Step 2.2: Create an IAM Role (execution role) (Amazon S3 Events) (p. 49)

Step 2.3: Upload the Deployment Package and Test (Amazon S3 Events) (p. 49)

Step 2.1: Create a Lambda Function Deployment Package

The deployment package is a zip file containing your function code and dependencies. So you will first create a directory to save the JavaScript function code and dependencies. After you complete the steps, you will have the following folder structure:

```
CreateThumbnail.js
/node_modules/gm
/node_modules/async
```

You will then zip the directory content, which is your Lambda function deployment package.

To create a Lambda function deployment package

1. Create a folder (`examplefolder`). After creating the folder, create a subfolder (node_modules) in it.
2. a. Install the Node.js platform. For more information, go to the Node.js website at http://nodejs.org/.
   b. Install dependencies. The code examples uses the following libraries.

      - AWS SDK for JavaScript in Node.js
      - gm, "GraphicsMagick for node.js"
      - Async utility module

      The AWS Lambda runtime already has the AWS SDK for JavaScript in Node.js. So you need only the other two. Open a command prompt, navigate to the `examplefolder`, and install the libraries using the `npm` command, which is part of Node.js.

      ```
      npm install async gm
      ```

3. Save example code to the folder. Open a text editor, and copy the following code.

```
// dependencies
var async = require('async');
var AWS = require('aws-sdk');
var gm = require('gm')
           .subClass({ imageMagick: true }); // Enable ImageMagick integra
tion.
var util = require('util');

// constants
var MAX_WIDTH  = 100;
var MAX_HEIGHT = 100;

// get reference to S3 client
var s3 = new AWS.S3();

exports.handler = function(event, context) {
 // Read options from the event.
 console.log("Reading options from event:\n", util.inspect(event, {depth:
5}));
 var srcBucket = event.Records[0].s3.bucket.name;
 // Object key may have spaces or unicode non-ASCII characters.
    var srcKey    =
    decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/g, " "));

 var dstBucket = srcBucket + "resized";
 var dstKey    = "resized-" + srcKey;

 // Sanity check: validate that source and destination are different buckets.

 if (srcBucket == dstBucket) {
  console.error("Destination bucket must not match source bucket.");
  return;
 }

 // Infer the image type.
 var typeMatch = srcKey.match(/\.([^.]*)$/);
 if (!typeMatch) {
  console.error('unable to infer image type for key ' + srcKey);
  return;
 }
 var imageType = typeMatch[1];
 if (imageType != "jpg" && imageType != "png") {
  console.log('skipping non-image ' + srcKey);
  return;
 }

 // Download the image from S3, transform, and upload to a different S3
bucket.
 async.waterfall([
  function download(next) {
   // Download the image from S3 into a buffer.
   s3.getObject({
     Bucket: srcBucket,
     Key: srcKey
    },
    next);
   },
  function transform(response, next) {
```

```
   gm(response.Body).size(function(err, size) {
    // Infer the scaling factor to avoid stretching the image unnaturally.
    var scalingFactor = Math.min(
     MAX_WIDTH / size.width,
     MAX_HEIGHT / size.height
    );
    var width  = scalingFactor * size.width;
    var height = scalingFactor * size.height;

    // Transform the image buffer in memory.
    this.resize(width, height)
     .toBuffer(imageType, function(err, buffer) {
      if (err) {
       next(err);
      } else {
       next(null, response.ContentType, buffer);
      }
     });
   });
  },
  function upload(contentType, data, next) {
   // Stream the transformed image to a different S3 bucket.
   s3.putObject({
     Bucket: dstBucket,
     Key: dstKey,
     Body: data,
     ContentType: contentType
    },
    next);
   }
  ], function (err) {
   if (err) {
    console.error(
     'Unable to resize ' + srcBucket + '/' + srcKey +
     ' and upload to ' + dstBucket + '/' + dstKey +
     ' due to an error: ' + err
    );
   } else {
    console.log(
     'Successfully resized ' + srcBucket + '/' + srcKey +
     ' and uploaded to ' + dstBucket + '/' + dstKey
    );
   }

   context.done();
  }
 );
};
```

4. Review the preceding code and note the following:

   - The function knows the source bucket name and the key name of the object from the event data it receives as parameters. If the object is a .jpg, the code creates a thumbnail and saves it to the target bucket.
   - The code assumes the destination bucket exists and its name is a concatenation of the source bucket name followed by the string "resized". For example, if the source bucket identified in the

event data is "examplebucket", the code assumes you have an "examplebucketresized" destination bucket.
- For the thumbnail it creates, the code derives its key name as the concatenation of the string "resized-" followed by the source object key name. For example, if the source object key is "sample.jpg", the code creates a thumbnail object that has the key "resized-sample.jpg".

5. Save the file as `CreateThumbnail.js` in `examplefolder`.
6. Zip the folder content as `CreateThumbnail.zip`.

   Important
   You zip the folder content, not the folder itself.

   This is your Lambda function deployment package.

Next Step

Step 2.2: Create an IAM Role (execution role) (Amazon S3 Events) (p. 49)

Step 2.2: Create an IAM Role (execution role) (Amazon S3 Events)

In this section, you create an IAM role using the following predefined role type and access policy:

- AWS service role of the "AWS Lambda" type. This role grants AWS Lambda permission to assume the role.
- "`AWSLambdaExecute`" access policy that you attach to the role.

For more information about IAM roles, go to Roles (Delegation and Federation) in *Using IAM*. Use the following procedure to create the IAM role.

To create an IAM role (execution role)

1. Sign in to the AWS Management Console.
2. Create an IAM role.

   For instructions on creating the role, go to Creating a Role for an AWS Service (AWS Management Console) in *IAM User Guide*. As you follow the steps to create a role, note the following:

   - In **Select Role Type**, click **AWS Service Roles**, and then select **AWS Lambda**. This will grant AWS Lambda service permission to assume the role.
   - In **Attach Policy** select `AWSLambdaExecute`.

3. Write down the role ARN. You will need it in the next step when you create your Lambda function.

Step 2.3: Upload the Deployment Package and Test (Amazon S3 Events)

In this section, you do the following:

- Create a Lambda function by uploading the deployment package.
- Test the Lambda function by invoking it manually, by passing sample Amazon S3 event data as parameter.

Step 2.3.1: Create a Lambda Function (Upload the Deployment Package)

1. At the command prompt, run the following Lambda CLI `create-function` command using the *adminuser* `profile`.

   You will need to update the command by providing the .zip file path and the execution role ARN.

```
$ aws lambda create-function \
--region us-west-2 \
--function-name CreateThumbnail \
--zip-file fileb://file-path/CreateThumbnail.zip \
--role role-arn \
--handler CreateThumbnail.handler \
--runtime nodejs \
--profile adminuser \
--timeout 10 \
--memory-size 1024
```

   Note that if you want to you can upload the .zip file to an Amazon S3 bucket in the same AWS region, and then specify the bucket and object name in the preceding command. You will need to replace the `--zip-file` parameter by the `--code` parameter as shown:

```
--code S3Bucket=bucket-name,S3Key=zip-file-object-key
```

2. Write down the function ARN. You will need this in the next section when you add notification configuration to your Amazon S3 bucket.
3. (Optional) The preceding command specifies a 10-second timeout value as the function configuration. Depending on the size of objects you upload, you might need to increase the timeout value using the following CLI command.

```
$ aws lambda update-function-configuration \
   --function-name CreateThumbnail  \
   --region us-west-2 \
   --timeout timeout-in-seconds \
   --profile adminuser
```

Step 2.3.2: Test Lambda Function (Invoke Manually)

Invoke the function manually using sample Amazon S3 event data.

1. Save the following Amazon S3 sample event data in a file, `input.txt`.

   You will need to update the JSON by providing your *`sourcebucket`* and a .jpg object key.

```
{
   "Records":[
      {
         "eventVersion":"2.0",
         "eventSource":"aws:s3",
         "awsRegion":"us-west-2",
         "eventTime":"1970-01-01T00:00:00.000Z",
         "eventName":"ObjectCreated:Put",
         "userIdentity":{
```

```
            "principalId":"AIDAJDPLRKLG7UEXAMPLE"
         },
         "requestParameters":{
            "sourceIPAddress":"127.0.0.1"
         },
         "responseElements":{
            "x-amz-request-id":"C3D13FE58DE4C810",
           "x-amz-id-2":"FMyUVURIY8/IgAtTv8xRjskZQpcIZ9KG4V5Wp6S7S/JRWeUWer
MUE5JgHvANOjpD"
         },
         "s3":{
            "s3SchemaVersion":"1.0",
            "configurationId":"testConfigRule",
            "bucket":{
               "name":"sourcebucket",
               "ownerIdentity":{
                  "principalId":"A3NL1KOZZKExample"
               },
               "arn":"arn:aws:s3:::sourcebucket"
            },
            "object":{
               "key":"HappyFace.jpg",
               "size":1024,
               "eTag":"d41d8cd98f00b204e9800998ecf8427e",
               "versionId":"096fKKXTRTtl3on89fVO.nfljtsv6qko"
            }
         }
      }
   ]
}
```

2. Run the following Lambda CLI `invoke` command to invoke the function. Note that:
   - The command requests asynchronous execution. You can optionally invoke it synchronously by specifying "RequestResponse" as the invocation-type parameter value.

```
$ aws lambda invoke \
--invocation-type Event \
--function-name CreateThumbnail \
--region us-west-2 \
--payload file://file-path/inputfile.txt \
--profile adminuser \
outputfile.txt
```

   Note
   You are able to invoke this function because you are using your own credentials to invoke your own function. In the next section, you configure Amazon S3 to invoke this function on your behalf which requires you to add a permission to the access policy associated with your Lambda function to grant Amazon S3 permission to invoke your function.

3. Verify results:
   - Verify the thumbnail was created in the target bucket.
   - You can monitor the activity of your Lambda function in the AWS Lambda console.
     - The AWS Lambda console shows a graphical representation of some of the CloudWatch metrics in the **Cloudwatch Metrics at a glance** section for your function.
     - For each graph, you can also click the **logs** link to view the CloudWatch logs directly.

Next Step

Step 3: Configure Amazon S3 to Publish Events (p. 52)

Step 3: Configure Amazon S3 to Publish Events

In this section, you add the remaining configuration so Amazon S3 can publish object-created events to AWS Lambda and invoke your Lambda function. You will do the following:

- Add permission to the Lambda function access policy to allow Amazon S3 to invoke the function.
- Add notification configuration to your source bucket — In the notification configuration, you provide:
  - Event type for which you want Amazon S3 to publish events. For this exercise, you will specify the `s3:ObjectCreated:*` event type so that Amazon S3 publishes events when objects are created.
  - Lambda function to invoke.

Step 3.1: Add Permission to the Lambda Function Access Policy

1. Run the following Lambda CLI `add-permission` command to grant Amazon S3 service principal ("s3.amazonaws.com") permission for the `lambda:InvokeFunction` action. Note that permission is granted to Amazon S3 to invoke the function only if the following conditions are met:

   - An object-created event is detected on a specific bucket.
   - The bucket is owned by a specific AWS account.

     If a bucket owner deletes a bucket, some other AWS account can create bucket with the same name. This condition ensures that only a specific AWS account can invoke your Lambda function.

```
$ aws lambda add-permission \
--function-name CreateThumbnail \
--region us-west-2 \
--statement-id some-unique-id \
--action "lambda:InvokeFunction" \
--principal s3.amazonaws.com \
--source-arn arn:aws:s3:::sourcebucket \
--source-account bucket-owner-account-id \
--profile adminuser
```

2. Verify the access policy of your function by calling the CLI `get-policy` command.

```
$ aws lambda get-policy \
--function-name function-name \
--profile adminuser
```

Step 3.2: Configure a Notification on the Bucket

Add notification configuration on the source bucket to request Amazon S3 to publish object-created events to Lambda. In the configuration, you specify the following:

- Event type — For this exercise, select the "ObjectCreated (All)" Amazon S3 event type.
- Lambda function — This is your Lambda function Amazon S3 will invoke.

For instructions on adding notification configuration to a bucket, go to Enabling Event Notifications in the *Amazon Simple Storage Service Console User Guide*.

Step 3.3: Test the Setup

You are all done! Now *adminuser* can test the setup as follows:

1. Upload .jpg or .png objects to the source bucket using the Amazon S3 console.
2. Verify that the thumbnail was created in the target bucket by the `CreateThumbnail` function.
3. The *adminuser* user can also verify the logs reported by Amazon CloudWatch Logs.

   You can monitor the activity of your Lambda function in the AWS Lambda console. For example, choose the logs link in the console to view logs, including logs your function wrote to CloudWatch Logs.

AWS Lambda Walkthrough 3: Processing Events from Amazon DynamoDB Streams Using the AWS CLI (Node.js)

Scenario

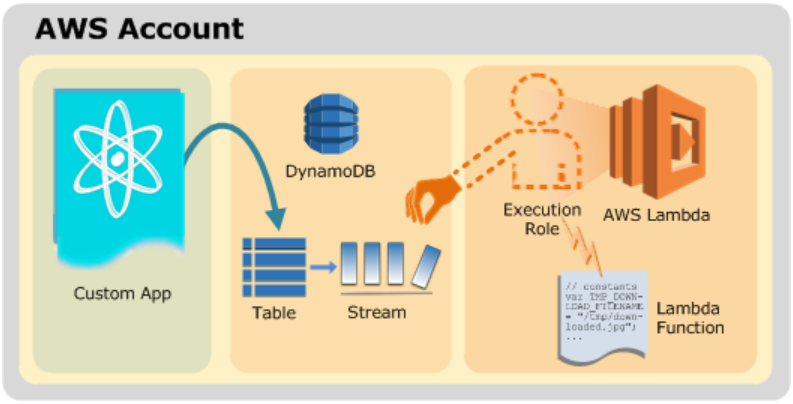
You can use Lambda functions as Triggers for your DynamoDB table. Triggers are custom actions you take in response to updates made to the DynamoDB table. To create a Trigger, first you enable DynamoDB Streams for your table. Then, you write a Lambda function to process the updates published to the stream. Lastly, you create an event source mapping in AWS Lambda to associate your Lambda function with the stream.

In this walkthrough, you create a simple Lambda function in which the `handler` receives event data as parameter and processes it. For illustration, the `handler` will log some of the information to Amazon CloudWatch.

Implementation Summary

This is an example of the "pull" model (see AWS Lambda: How it Works (p. 3)) where AWS Lambda polls the Amazon DynamoDB stream and invokes your Lambda function when it detects new data on the stream. In the "pull" model, AWS Lambda both invokes and executes the Lambda function.

The following diagram illustrates the application workflow:



In this walkthrough, you will do the following:

- Create a Lambda function to process Amazon DynamoDB events.
- Invoke a Lambda function manually by using sample Amazon DynamoDB event data.
- Create a stream-enabled DynamoDB table.

  Using the DynamoDB console, you will create a table with a stream.
- Create an event source mapping in AWS Lambda associating the stream and your Lambda function.

  As soon as you create the event source mapping, AWS Lambda starts polling the stream.
- Test the setup.

  As you create, update, and delete items from the table, Amazon DynamoDB writes records to the stream. AWS Lambda will detect the new records as it polls the stream and execute your Lambda function on your behalf. You then verify AWS Lambda executed your Lambda function on your behalf.

Important
Both the Lambda function and the Amazon DynamoDB stream must be in the same AWS region. This exercise assumes the US East (N. Virginia) region (us-east-1).

Next Step

Step 1: Prepare for the Walkthrough (DynamoDB Stream Events) (p. 54)

Step 1: Prepare for the Walkthrough (DynamoDB Stream Events)

In this section, you do the following:

- If you don't have an administrator user in your account, create it. For instructions, see Set Up an AWS Account and Create an Administrator User (p. 12).
- Set up the AWS CLI. For instructions, see Step 1: Prepare for the Walkthrough (p. 36).

  You will use AWS CLI to perform the AWS Lambda activities, such as create a Lambda function, initially invoke it manually, and add an event source mapping.

Next Step

Step 2: Create a Lambda Function and Invoke it Manually Using Sample Event Data (DynamoDB Stream Events) (p. 54)

Step 2: Create a Lambda Function and Invoke it Manually Using Sample Event Data (DynamoDB Stream Events)

Topics

In this section, you create a Lambda function and manually invoke it by passing a sample DynamoDB event.

Step 2.1: Create a Lambda Function

To create a Lambda function you need to first create the following:

- A deployment package (a .zip file) containing your code and dependencies.

  After you upload the deployment package to AWS Lambda we refer it as your Lambda function. The code for this exercise is a Node.js example code, and there are no dependencies.
- An IAM role (execution role).

  At the time you create your Lambda function you specify an IAM role that the AWS Lambda service can assume to execute the function on your behalf.

  You must grant this execution role necessary permissions. For example AWS Lambda will need permission for Amazon DynamoDB actions so it can poll the stream and read records from the stream. In the "pull" model you must also grant AWS Lambda permission to invoke your Lambda function. The example Lambda function writes some of the event data to Amazon CloudWatch, so it will need permissions for necessary Amazon CloudWatch actions.

  For more information, see Execution Permissions (p. 8).

You provide both the deployment package and the IAM role at the time of creating a Lambda function. You can also specify other configuration information such as the function name, memory size, runtime environment (nodejs) to use, and the handler. For more information about these parameters, see CreateFunction (p. 160).

After creating the Lambda function you will invoke it using sample Amazon DynamoDB event data.

Step 2.1.1: Create a Lambda Function Deployment Package

Follow the instructions to create your AWS Lambda function deployment package.

1. Open a text editor, and copy the following code.

```
console.log('Loading function');

exports.handler = function(event, context) {
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(function(record) {
        console.log(record.eventID);
        console.log(record.eventName);
        console.log('DynamoDB Record: %j', record.dynamodb);
    });
    context.succeed("Successfully processed " + event.Records.length + "
records.");
};
```

2. Save the file as `ProcessDynamoDBStream.js`.
3. Zip the `ProcessDynamoDBStream.js` file as `ProcessDynamoDBStream.zip`.

Step 2.1.2: Create an IAM Role (execution role)

In this section you create an IAM role using the following predefined role type and access policy:

- Role Type: AWS service role of the "AWS Lambda" type.

This role grants AWS Lambda permission to assume the role. So when you specify this role when creating your Lambda function, AWS Lambda will be able to assume this role and execute your Lambda function on your behalf.

- Access Policy: You will grant permission to the role by attaching a predefined access policy ("AWSLambdaDynamoDBExecutionRole").

  The policy grants necessary permission your Lambda function needs when it executes.

Use the following procedure to create the IAM role.

1. Sign in to the AWS Management Console.
2. Create an IAM role.

   For instructions on creating the role, go to Creating a Role for an AWS Service (AWS Management Console) in *IAM User Guide*. As you follow the steps to create a role, note the following:

   - In **Select Role Type**, click **AWS Service Roles**, and then select **AWS Lambda**.
   - In **Attach Policy** select `AWSLambdaDynamoDBExecutionRole`.

3. Write down the role ARN. You will need the ARN in the next step when you create Lambda function.

For more information about IAM roles, go to Roles (Delegation and Federation) in *Using IAM*.

Step 2.1.3: Create Lambda Function

Execute the following Lambda CLI `create-function` command to create a Lambda function. You provide the deployment package and IAM role ARN as parameters.

```
$ aws lambda create-function \
--region us-east-1 \
--function-name ProcessDynamoDBStream \
--zip-file fileb://file-path/ProcessDynamoDBStream.zip \
--role role-arn \
--handler ProcessDynamoDBStream.handler \
--runtime nodejs \
--profile adminuser
```

For more information, see CreateFunction (p. 160). AWS Lambda creates the function and return function configuration information.

Note
You can upload the .zip file to an Amazon S3 bucket in the same AWS region where you are creating the Lambda function, and then specify the bucket and object name in the `create-function` command. You will need to replace the `--zip-file` parameter by the `--code` parameter as shown:

```
--code S3Bucket=bucket-name,S3Key=zip-file-object-key
```

Step 2.2: Invoke Lambda Function Manually

In this section you invoke your Lambda function manually using the `invoke` AWS Lambda CLI command and the following sample DynamoDB event.

1. Save the following JSON in a file, `input.txt`.

```
{
   "Records":[
      {
         "eventID":"1",
         "eventName":"INSERT",
         "eventVersion":"1.0",
         "eventSource":"aws:dynamodb",
         "awsRegion":"us-east-1",
         "dynamodb":{
            "Keys":{
               "Id":{
                  "N":"101"
               }
            },
            "NewImage":{
               "Message":{
                  "S":"New item!"
               },
               "Id":{
                  "N":"101"
               }
            },
            "SequenceNumber":"111",
            "SizeBytes":26,
            "StreamViewType":"NEW_AND_OLD_IMAGES"
         },
         "eventSourceARN":"stream-ARN"
      },
      {
         "eventID":"2",
         "eventName":"MODIFY",
         "eventVersion":"1.0",
         "eventSource":"aws:dynamodb",
         "awsRegion":"us-east-1",
         "dynamodb":{
            "Keys":{
               "Id":{
                  "N":"101"
               }
            },
            "NewImage":{
               "Message":{
                  "S":"This item has changed"
               },
               "Id":{
                  "N":"101"
               }
            },
            "OldImage":{
               "Message":{
                  "S":"New item!"
               },
               "Id":{
                  "N":"101"
               }
            },
            "SequenceNumber":"222",
            "SizeBytes":59,
```

```
            "StreamViewType":"NEW_AND_OLD_IMAGES"
         },
         "eventSourceARN":"stream-ARN"
      },
      {
         "eventID":"3",
         "eventName":"REMOVE",
         "eventVersion":"1.0",
         "eventSource":"aws:dynamodb",
         "awsRegion":"us-east-1",
         "dynamodb":{
            "Keys":{
               "Id":{
                  "N":"101"
               }
            },
            "OldImage":{
               "Message":{
                  "S":"This item has changed"
               },
               "Id":{
                  "N":"101"
               }
            },
            "SequenceNumber":"333",
            "SizeBytes":38,
            "StreamViewType":"NEW_AND_OLD_IMAGES"
         },
         "eventSourceARN":"stream-ARN"
      }
   ]
}
```

2. Execute the following `invoke` command.

```
$ aws lambda invoke \
--invocation-type RequestResponse \
--function-name ProcessDynamoDBStream \
--region us-east-1 \
--payload file://file-path/input.txt \
--profile adminuser \
outputfile.txt
```

   Note the `invoke` command specifies the "RequestResponse" as the invocation type which requests synchronous execution. For more information, see Invoke (p. 178). The function returns the string message (message in the `context.succeed()` in the code) in the response body.

3. Verify the following output in the `outputfile.txt` file.

```
"Successfully processed 3 records."
```

   You can monitor the activity of your Lambda function in the AWS Lambda console.

   - The AWS Lambda console shows a graphical representation of some of the CloudWatch metrics in the **Cloudwatch Metrics at a glance** section for your function. Sign in to the AWS Management Console at https://console.aws.amazon.com/.

- For each graph you can also click the **logs** link to view the CloudWatch logs directly.

Next Step

Step 3: Add an Event Source (DynamoDB Streams) and Test (p. 59)

Step 3: Add an Event Source (DynamoDB Streams) and Test

In this section, you do the following:

- Create an Amazon DynamoDB table with a stream enabled.
- Create an event source mapping in AWS Lambda.

  This event source mapping will associate the DynamoDB stream with your Lambda function. After you create this event source mapping, AWS Lambda will start polling the stream.
- Test the setup.

  As you perform table updates, DynamoDB will write event records to the stream. AWS Lambda that is polling the stream will detect new records in the stream and execute your Lambda function on your behalf, by passing events to the function.

Step 3.1: Create a DynamoDB Table with a Stream Enabled

Follow the procedure to create a table with a stream:

1. Sign in to the AWS Management console at http://aws.amazon.com/console/.
2. In the DynamoDB console, create a table with streams enabled. Make sure you have the US East (N. Virginia) region selected before you create the table.

   Important
   You must create a DynamoDB table in the same region that you have a Lambda function created. This exercise assume the US East (N. Virginia) region. In addition, both the table and the Lambda functions must belong to the same AWS account.
3. Write down the stream ARN. You will need this in the next section when you associate the stream with your Lambda function.

Step 3.2: Create Event Source Mapping in AWS Lambda

In this section you associate your DynamoDB stream with your Lambda function by creating a event source mapping in AWS Lambda.

Run the following AWS CLI `create-event-source-mapping` command. After the command executes, note down the UUID. You'll need this UUID to refer to the event source mapping in any commands, for example, when deleting the event source mapping.

```
$ aws lambda create-event-source-mapping \
--region us-east-1 \
--function-name ProcessDynamoDBStream \
--event-source DynamoDB-stream-arn \
--batch-size 100 \
--starting-position TRIM_HORIZON \
--profile adminuser
```

Note
This creates a mapping between the specified DynamoDB stream and the Lambda function. You can associate a DynamoDB stream with multiple Lambda functions, and associate the same Lambda function with multiple streams. However, the Lambda functions will share the read throughput for the stream they share.

You can get the list of event source mappings.

```
$ aws lambda list-event-source-mappings \
--region us-east-1 \
--function-name ProcessDynamoDBStream \
--event-source DynamoDB-stream-arn \
--profile adminuser
```

The list returns all the event source mappings you created, and for each mapping it shows (among other things) the `LastProcessingResult`. This field is used to provide an informative message if there are any problems. Values such as "No records processed" (indicating either AWS Lambda has not started polling or there are no records in the stream), and "OK" (indicating AWS Lambda successfully read records from the stream and invoked your Lambda function) indicate there no issues. Otherwise, you will get an appropriate message.

Step 3.3: Test the Setup

You are all done! Now *adminuser* can test the setup as follows:

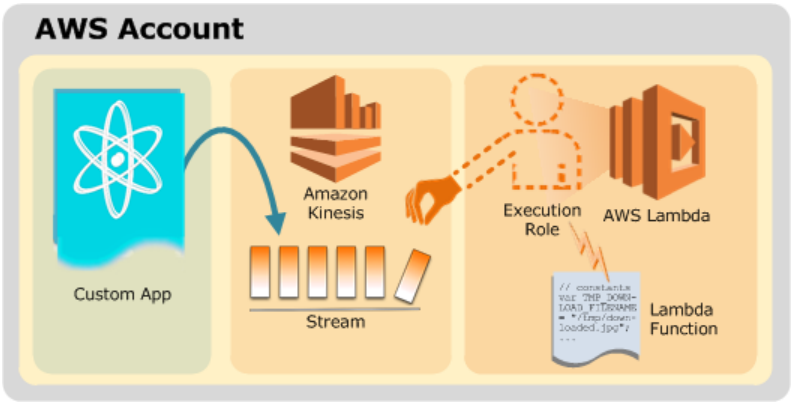
1. In the DynamoDB console, add, update, delete items to the table. DynamoDB will write records of these actions to the stream.
2. AWS Lambda polls the stream and when it detects updates to the stream, it will invoke your Lambda function by passing in event data it found in the stream.
3. Your function executes and creates logs in Amazon CloudWatch. The *adminuser* can also verify the logs reported in the Amazon CloudWatch console.

AWS Lambda Walkthrough 4: Processing Events from an Amazon Kinesis Stream Using the AWS CLI (Node.js)

In this walkthrough, you create a Lambda function to consume events from an Amazon Kinesis stream. The Lambda function is simple: it reads incoming event data and logs some of the information to Amazon CloudWatch.

This is an example of the "pull" model (see AWS Lambda: How it Works (p. 3)) where AWS Lambda polls the Amazon Kinesis stream and invokes your Lambda function when it detects new data on the stream. That is, in the pull model, AWS Lambda both invokes and executes the Lambda function.

The following diagram illustrates the application workflow:



In this walkthrough, you will do the following:

- Create a Lambda function to process Amazon Kinesis events
- Invoke a Lambda function manually using sample Amazon Kinesis event data
- Create an Amazon Amazon Kinesis stream
- Add an event source in AWS Lambda associating the stream and your Lambda function.

  As soon as you add the event source, AWS Lambda starts polling the stream.
- Test the setup

  You will then add a sample event record to the Amazon Kinesis stream and verify AWS Lambda executed your Lambda function on your behalf.

Important
Both the Lambda function and the Amazon Kinesis stream must be in the same AWS region. This exercise assumes the `us-west-2` region.

Note
In this walkthrough, you use the AWS Command Line Interface to perform AWS Lambda operations such as the create and invoke functions. This example is similar to the getting started example (see Getting Started 3: Handling Amazon Kinesis Events Using the AWS Lambda Console (Node.js) (p. 23)). The difference is that the Getting Started exercise provides a console-based experience. The console does many things for you, simplifying your experience. When you use the CLI, you get the raw experience of making the API calls, which can help you familiarize yourself with the AWS Lambda operations. In addition to creating and invoking Lambda function, you will explore other Lambda APIs.

Next Step

Step 1: Prepare for the Walkthrough (Amazon Kinesis Stream Events) (p. 61)

Step 1: Prepare for the Walkthrough (Amazon Kinesis Stream Events)

In this section, you do the following:

- If you don't have an administrator user in your account, create one. For instructions, see Set Up an AWS Account and Create an Administrator User (p. 12).
- Set up the AWS CLI. For instructions, see Step 1: Prepare for the Walkthrough (p. 36).

You will use AWS CLI to perform the AWS Lambda activities, such as create a Lambda function, initially invoke it manually, and add an event source.

Next Step

Step 2: Create a Lambda Function and Invoke it Manually Using Sample Event Data (Amazon Kinesis Stream Events) (p. 62)

Step 2: Create a Lambda Function and Invoke it Manually Using Sample Event Data (Amazon Kinesis Stream Events)

Topics

In this section, you create a Lambda function and manually invoke it by passing sample Amazon Kinesis event.

Step 2.1: Create a Lambda Function

To create a Lambda function, you need to first create the following:

- A deployment package (a .zip file) containing your code and dependencies.

  After you upload the deployment package to AWS Lambda, we refer it as your Lambda function. The code for this exercise is a Node.js example, and there are no dependencies.
- An IAM role (execution role).

  At the time you create your Lambda function, you specify an IAM role that AWS Lambda can assume to execute the function on your behalf.

  You must grant this execution role the necessary permissions. For example, AWS Lambda will need permission for Amazon Kinesis actions so it can poll the stream and read records from the stream. In the pull model, you must also grant AWS Lambda permission to invoke your Lambda function. The example Lambda function writes some of the event data to Amazon CloudWatch so your function will need permissions for the necessary Amazon CloudWatch actions.

  For more information, see Execution Permissions (p. 8)).

You provide both the deployment package and the IAM role when you create a Lambda function. You can also specify other configuration information such as the function name, memory size, runtime environment (nodejs) to use, and the handler. For more information about these parameters, see CreateFunction (p. 160).

After creating the Lambda function, you will invoke it using sample Amazon Kinesis event data.

Step 2.1.1: Create a Lambda Function Deployment Package

Follow the instructions to create AWS Lambda function deployment package.

1. Open a text editor, and copy the following code.

```
console.log('Loading function');

exports.handler = function(event, context) {
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(function(record) {
        // Kinesis data is base64 encoded so decode here
       payload = new Buffer(record.kinesis.data, 'base64').toString('ascii');

        console.log('Decoded payload:', payload);
    });
    context.succeed();
};
```

2. Save the file as `ProcessKinesisRecords.js`.
3. Zip the `ProcessKinesisRecords.js` file as `ProcessKinesisRecords.zip`.

Step 2.1.2: Create an IAM Role (execution role)

In this section, you create an IAM role using the following predefined role type and access policy:

- AWS service role of the "AWS Lambda" type. This role grants AWS Lambda permission to assume the role.
- "`AWSLambdaKinesisExecutionRole`" access policy that you attach to the role.

For more information about IAM roles, go to Roles (Delegation and Federation) in *Using IAM*. Use the following procedure to create the IAM role.

To create an IAM role (*executionrole*)

1. Sign in to the AWS Management Console.
2. Create an IAM role.

   For instructions on creating the role, go to Creating a Role for an AWS Service (AWS Management Console) in *IAM User Guide*. As you follow the steps to create a role, note the following:

   - In **Select Role Type**, click **AWS Service Roles**, and then select **AWS Lambda**. This will grant AWS Lambda service permission to assume the role.
   - In **Attach Policy** select `AWSLambdaKinesisExecutionRole`.

     .

3. Write down the role ARN. You will need it in the next step when you create your Lambda function.

Step 2.1.3: Create a Lambda Function

Execute the following Lambda CLI `create-function` command to create a Lambda function. You provide the deployment package and IAM role ARN as parameters.

```
$ aws lambda create-function \
--region us-west-2 \
--function-name ProcessKinesisRecords  \
--zip-file fileb://file-path/ProcessKinesisRecords.zip \
--role execution-role-arn  \
```

```
--handler ProcessKinesisRecords.handler \
--runtime nodejs \
--profile adminuser
```

Note that if you want you can upload the .zip file to an Amazon S3 bucket in the same AWS region, and then specify the bucket and object name in the preceding command. You will need to replace the `--zip-file` parameter by the `--code` parameter as shown:

```
--code S3Bucket=bucket-name,S3Key=zip-file-object-key
```

For more information, see CreateFunction (p. 160). AWS Lambda creates the function and returns function configuration information as shown in the following example:

```
{
    "FunctionName": "ProcessKinesisRecords",
    "CodeSize": 412,
    "MemorySize": 128,
    "FunctionArn": "arn:aws:lambda:us-west-2:account-id:function:ProcessKines
isRecords",
    "Handler": "ProcessKinesisRecords.handler",
    "Role": "arn:aws:iam::account-id:role/kinesis-lambda-role",
    "Timeout": 3,
    "LastModified": "2015-04-02T01:20:42.355+0000",
    "Runtime": "nodejs",
    "Description": ""
}
```

Step 2.2: Invoke Your Lambda Function Manually

In this section, you invoke your Lambda function manually using the `invoke` CLI command.

Save the following JSON in a file, `input.txt`.

```
{
    "Records": [
        {
            "kinesis": {
                "partitionKey": "partitionKey-3",
                "kinesisSchemaVersion": "1.0",
                "data": "SGVsbG8sIHRoaXMgaXMgYSB0ZXN0IDEyMy4=",
                "sequenceNumber":
"49545115243490985018280067714973144582180062593244200961"
            },
            "eventSource": "aws:kinesis",
            "eventID": "shardId-
000000000000:49545115243490985018280067714973144582180062593244200961",
            "invokeIdentityArn": "arn:aws:iam::059493405231:role/testLEBRole",

            "eventVersion": "1.0",
            "eventName": "aws:kinesis:record",
          "eventSourceARN": "arn:aws:kinesis:us-west-2:35667example:stream/ex
amplestream",
            "awsRegion": "us-west-2"
        }
```

```
    ]
}
```

Execute the following `invoke` command.

```
$ aws lambda  invoke \
--invocation-type Event \
--function-name ProcessKinesisRecords \
--region us-west-2 \
--payload file://file-path/input.txt \
--profile adminuser
outputfile.txt
```

Note that if you request synchronous execution ("RequestResponse" as the invocation type), function returns the string message (message in the `context.succeed()` in the code) in the response body. In the preceding example it will be saved in `outputfile.txt`.

```
"Hello, this is a test 123."
```

Note
The Amazon Kinesis stream and Lambda function must be in the same AWS account.

Next Step

Step 3: Add an AWS Lambda Event Source and Test (Amazon Kinesis Stream Events) (p. 65)

Step 3: Add an AWS Lambda Event Source and Test (Amazon Kinesis Stream Events)

Topics

In this section, you create an Amazon Kinesis stream and add an event source in AWS Lambda to associate the stream with your Lambda function. After you create an event source, AWS Lambda will start polling the stream. You will then test the setup by adding events to the stream and verify that AWS Lambda executed your Lambda function on your behalf.

Step 3.1: Create an Amazon Kinesis Stream

Use the following Amazon Kinesis `create-stream` CLI command to create a stream.

```
$ aws kinesis create-stream \
--stream-name examplestream \
--shard-count 1 \
--region us-west-2 \
--profile adminuser
```

Run the following Amazon Kinesis `describe-stream` CLI command to get the stream ARN.

```
$ aws kinesis describe-stream \
--stream-name examplestream \
--region us-west-2 \
--profile adminuser
```

You need the stream ARN in the next step to associate the stream with your Lambda function. The stream is of the form:

```
arn:aws:kinesis:aws-region:account-id:stream/stream-name
```

Step 3.2: Add an Event Source in AWS Lambda

Run the following AWS CLI `add-event-source` command. After the command executes, note down the UUID. You'll need this UUID to refer to the event source in any commands, for example, when deleting the event source.

```
$ aws lambda create-event-source-mapping \
--region us-west-2 \
--function-name ProcessKinesisRecords \
--event-source  kinesis-stream-arn \
--batch-size 100 \
--starting-position TRIM_HORIZON \
--profile adminuser
```

Note
This creates a mapping between the specified Amazon Kinesis stream and the Lambda function. You can associate an Amazon Kinesis stream with only one Lambda function. If you associate another function with the same stream, it replaces the previous mapping.

You can get a list of event source mappings.

```
$ aws lambda list-event-source-mappings \
--region us-west-2 \
--function-name ProcessKinesisRecords \
--event-source kinesis-stream-arn \
--debug
```

In the response, you can verify the status value is "enabled".

Step 3.3: Test the Setup

You are all done! Now *adminuser* can test the setup as follows:

1. Using the following AWS CLI command, add event records to your Amazon Kinesis stream. The `--data` value is a base64-encoded value of the "Hello, this is a test." string. You can run the same command more than once to add multiple records to the stream.

   ```
   $ aws kinesis put-record \
   --stream-name examplestream \
   --data "This is a test. final" \
   --partition-key shardId-000000000000 \
   --region us-west-2 \
   --profile adminuser
   ```

2. AWS Lambda polls the stream and when it detects updates to the stream, it will invoke your Lambda function by passing in event data it found in the stream.

   AWS Lambda assumes the execution role to poll the stream. You have granted the role permissions for necessary Amazon Kinesis actions, and therefore AWS Lambda can poll the stream and read events from the stream.
3. Your function executes and adds logs to the log group that corresponds to the Lambda function in Amazon CloudWatch.

   The *adminuser* can also verify the logs reported in the Amazon CloudWatch console. Make sure you are checking for logs in the same AWS region where you created the Lambda function.

AWS Lambda Walkthrough 5: Handling AWS CloudTrail Events Using the AWS CLI (Node.js)

Scenario

Suppose you have turned on AWS CloudTrail for your AWS account to maintain records (logs) of AWS API calls made on your account. As API calls are made in your account, CloudTrail writes logs to an Amazon S3 bucket you configured. You want Amazon S3 to publish the "object created" events to AWS Lambda and invoke your Lambda function, as CloudTrail creates log objects.

When Amazon S3 invokes your Lambda function, it will pass an S3 event identifying, among other things, the bucket name and the new object key that CloudTrail created. So your Lambda function can read the log object, and it will know the API calls that were reported in the log.

You want your Lambda function to notify you via email if the log reports a specific API was called. Each CloudTrail is a JSON object with one or more event records. Each record, among other things, provides `eventSource` and `eventName`.

```
{
    "Records":[

        {
            "eventVersion":"1.02",
            "userIdentity":{
                ...
            },
            "eventTime":"2014-12-16T19:17:43Z",
            "eventSource":"sns.amazonaws.com",
            "eventName":"CreateTopic",
            "awsRegion":"us-west-2",
            "sourceIPAddress":"72.21.198.64",
             ...
         },
         {
            ...
         },
         ...
}
```

Your Lambda function will parse the log for records with specific `eventSource` ("sns.amazonaws.com") and `eventName` ("CreateTopic"). If found, it will publish the event to your Amazon SNS topic, which you will configure to send email.

Implementation Summary

Upon completing this walkthrough, you will have Amazon S3, AWS Lambda, Amazon SNS, and AWS Identity and Access Management (IAM) resources in your account:

> **Note**
> The walkthrough assumes you create these resources in the `us-west-2` region.

In Lambda:

- A Lambda function – The function first reads incoming S3 event data so it knows where the CloudTrail log object is created. It will then read that object and process as explained in the preceding section.
- An access policy – In the Lambda function's access policy, you will add a permission to allow Amazon S3 to invoke the Lambda function.

In IAM:

- Administrator user – Called *adminuser*.

  You use the *adminuser* credentials to performs steps in this walkthrough.
- An IAM role (*executionrole*) – When you create this role, you will trust AWS Lambda to assume this role. You will also grant sufficient permissions that your Lambda function needs—for example, permissions for Amazon S3 actions to read objects from a bucket, permission for Amazon SNS actions to publish events, and permissions for CloudWatch actions to write logs.

In Amazon S3:

- A bucket – We refer to the bucket as *`examplebucket`*. When you turn the trail on in the CloudTrail console, you specify this bucket for CloudTrail to save the logs.
- Configure notification on *`examplebucket`*.

  You will add notification configuration to your bucket to request Amazon S3 to publish object-created events to Lambda, by invoking your Lambda function. For more information about the Amazon S3 notification feature, go to Setting Up Notification of Bucket Events.
- Sample CloudTrail log object, `ExampleCloudTrailLog.json`, in *`examplebucket`* bucket.

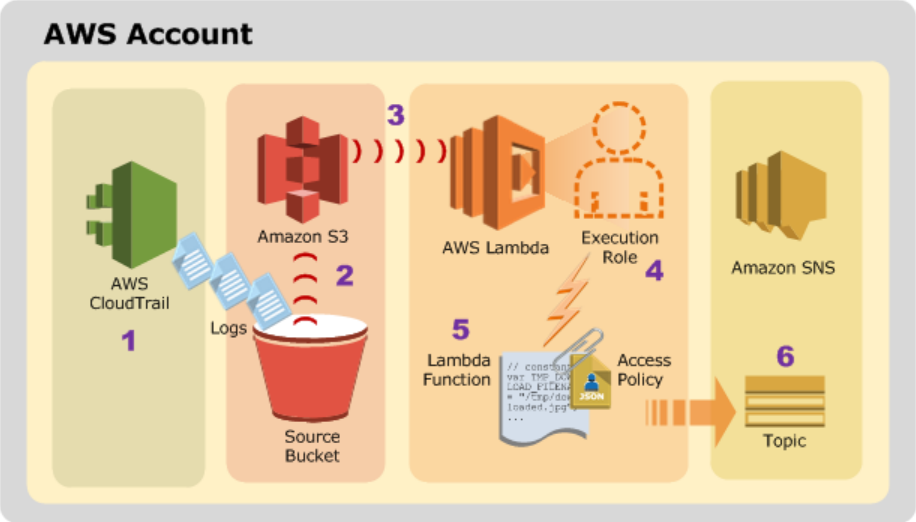
  You will use a sample Amazon S3 event that will identify *`examplebucket`* and this object as a CloudTrail object. You will use this Amazon S3 object to first manually invoke your Lambda function. You Lambda function will then read the sample CloudTrail log object and send you email notifications via an SNS topic.

In Amazon SNS

- An SNS topic.

  You subscribe to this topic by specifying email as the protocol.

The following diagram illustrates the application flow:



1. AWS CloudTrail saves logs to an S3 bucket.
2. Amazon S3 detects the object-created event.
3. Amazon S3 publishes the `s3:ObjectCreated:*` event to AWS Lambda by invoking the Lambda function, per the bucket notification configuration.

   Because the Lambda function's access policy includes a permission for Amazon S3 to invoke the function, Amazon S3 will be able to invoke the function.
4. Lambda assumes the execution role and executes the function.
5. The Lambda function first reads the Amazon S3 event it receives as a parameter and determines where the CloudTrail object is. It then reads the CloudTrail object and processes log records in it.
6. If the log includes a record with specific `eventType` and `eventSource` values, it publishes the event to your Amazon SNS topic.

   In this walkthrough, you subscribe to the SNS topic using the email protocol, so you will get email notifications.

Now you are ready to try the steps.

First Walkthrough Step

Step 1: Prepare for the Walkthrough (AWS CloudTrail Events) (p. 69)

Step 1: Prepare for the Walkthrough (AWS CloudTrail Events)

In this section, you do the following (the example assumes you are creating your AWS resources—S3 bucket, SNS topic, and Lambda function—in the `us-west-2` region:

- Create an administrator user, *adminuser*. For instructions, see Set Up an AWS Account and Create an Administrator User (p. 12). If you already have an administrator user, you can skip this step.
- Set up the AWS CLI. If you followed the walkthroughs in order, you already have the AWS CLI setup. If not, follow the instructions provided in walkthrough 1. For more information, see Step 1: Prepare for the Walkthrough (p. 36).
- Turn CloudTrail on. For instructions, see the following section.
- Create an SNS topic. For instructions, see the following section.

Step 1.1: Turn on CloudTrail

In the AWS CloudTrail console, turn on the trail in your account by specifying `examplebucket` in the `us-west-2` region for CloudTrail to save logs. When configuring the trail, do not enable SNS notification.

For instructions, go to Creating and Updating Your Trail in the *AWS CloudTrail User Guide*.

Step 1.2: Create an SNS Topic and Subscribe to the Topic

Follow the procedure to create an SNS topic in the `us-west-2` region and subscribe to it by providing an email address as the endpoint.

To create and subscribe to a topic

1. Create an SNS topic.

   For instructions, go to Create a Topic in the *Amazon Simple Notification Service Developer Guide*.
2. Subscribe to the topic by providing an email address as the endpoint.

   For instructions, go to Subscribe to a Topic in the *Amazon Simple Notification Service Developer Guide*.
3. Note down the topic ARN. You will need the value in the following sections.

Next Step

Step 2: Create and Invoke a Lambda Function (AWS CloudTrail Events) (p. 70)

Step 2: Create and Invoke a Lambda Function (AWS CloudTrail Events)

In this section, the *adminuser* creates a function in AWS Lambda and invokes it manually, using the AWS CLI command, using sample Amazon S3 event data. For instructions, see the following sections:

Step 2.1: Create a Lambda Function Deployment Package (AWS CloudTrail Events) (p. 70)

Step 2.2: Create an IAM Role (execution role) (AWS CloudTrail Events) (p. 73)

Step 2.3: Create a Lambda Function (AWS CloudTrail Events) (p. 74)

Step 2.4: Invoke Your Lambda Function Manually (AWS CloudTrail Events) (p. 75)

After you complete the steps, you can add a notification configuration on the `examplebucket` and test the end-to-end experience. For instructions, see the following section.

Step 3: Configure Amazon S3 to Publish Events (AWS CloudTrail Events) (p. 78).

Step 2.1: Create a Lambda Function Deployment Package (AWS CloudTrail Events)

The code example is created using Node.js. You first creates a folder to save the following:

- An example JavaScript function
- Dependencies

After you complete the steps, you will have the following folder structure:

```
CloudTrailEventProcessing.js
/node_modules/async
```

You will then zip the folder content; the .zip file is the Lambda function deployment package.

To create the Lambda function deployment package

1. Install the Node.js platform. For more information, go to the Node.js website at http://nodejs.org/.
2. Create a folder (examplefolder). After creating the folder, create a subfolder (node_modules) in it.
3. Open a command prompt, navigate to the `examplefolder`, and install the following libraries using the `npm` command, which is part of Node.js.

   - async (Async utility module)

   ```
   npm install async
   ```

4. Open a text editor, and copy the following code.

   ```
   var aws  = require('aws-sdk');
   var zlib = require('zlib');
   var async = require('async');

   var EVENT_SOURCE_TO_TRACK = /sns.amazonaws.com/;
   var EVENT_NAME_TO_TRACK   = /CreateTopic/;
   var DEFAULT_SNS_REGION  = 'us-west-2';
   var SNS_TOPIC_ARN       = 'SNS Topic ARN';

   var s3 = new aws.S3();
   var sns = new aws.SNS({
       apiVersion: '2010-03-31',
       region: DEFAULT_SNS_REGION
   });

   exports.handler = function(event, context) {
       var srcBucket = event.Records[0].s3.bucket.name;
       var srcKey = event.Records[0].s3.object.key;

       async.waterfall([
           function fetchLogFromS3(next){
               console.log('Fetching compressed log from S3...');
               s3.getObject({
                  Bucket: srcBucket,
                  Key: srcKey
               },
               next);
           },
           function uncompressLog(response, next){
               console.log("Uncompressing log...");
               zlib.gunzip(response.Body, next);
           },
           function publishNotifications(jsonBuffer, next) {
               console.log('Filtering log...');
               var json = jsonBuffer.toString();
               console.log('CloudTrail JSON from S3:', json);
   ```

```
            var records;
            try {
                records = JSON.parse(json);
            } catch (err) {
                next('Unable to parse CloudTrail JSON: ' + err);
                return;
            }
            var matchingRecords = records
                .Records
                .filter(function(record) {
                    return record.eventSource.match(EVENT_SOURCE_TO_TRACK)
                        && record.eventName.match(EVENT_NAME_TO_TRACK);
                });

           console.log('Publishing ' + matchingRecords.length + ' notific
ation(s) in parallel...');
            async.each(
                matchingRecords,
                function(record, publishComplete) {
                    console.log('Publishing notification: ', record);
                    sns.publish({
                        Message:
                            'Alert... SNS topic created: \n TopicARN=' +
record.responseElements.topicArn + '\n\n' +
                            JSON.stringify(record),
                        TopicArn: SNS_TOPIC_ARN
                    }, publishComplete);
                },
                next
            );
        }
    ], function (err) {
        if (err) {
            console.error('Failed to publish notifications: ', err);
        } else {
            console.log('Successfully published all notifications.');
        }
        context.done(err);
    });
};
```

5. Update the code by providing your SNS topic ARN.
6. Save the file as `CloudTrailEventProcessing.js` in `examplefolder`.
7. Zip the folder content as `CloudTrailEventProcessing.zip`.

   Note
   You zip the folder content, not the folder itself.

   This is your Lambda function deployment package.

Next Step

Step 2.2: Create an IAM Role (execution role) (AWS CloudTrail Events) (p. 73)

Step 2.2: Create an IAM Role (execution role) (AWS CloudTrail Events)

Now the *adminuser* is ready to create a Lambda function by uploading the deployment package to Lambda that you created in the previous step. But at the time of creating the function, *adminuser* will need to specify an IAM role that Lambda can assume. This role will have permissions the Lambda function needs to access your AWS resources. So let's first create an IAM role, *executionrole*. For more information about the *executionrole*, see Execution Permissions (p. 8).

Each IAM role has two policies. For the *executionrole*, you will use the following two policies:

- Trust policy specifying that AWS Lambda account principal who can assume the role.
- Access policy defining permissions for this role.

  The role will have permission for Amazon S3 actions to perform operations on the *examplebucket*, CloudWatch actions so the function can write application logs to CloudWatch, and Amazon SNS actions so the function can publish events to your SNS topic.

To create an IAM role (*executionrole*)

1. Sign in to the AWS Management Console using the *adminuser* credentials.
2. Create a managed policy that you will attach to the IAM role.

   a. In the navigation pane of the IAM console, click **Policies**, and then click **Create Policy**.
   b. Next to **Copy an AWS Managed Policy**, click **Select**.
   c. Next to **AWSLambdaExecute**, click **Select**.
   d. Copy the following policy into the **Policy Document** replacing the existing policy, and then update the policy with the ARN of the Amazon SNS topic you created.

      Note the policy name because you will use it in the next step.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "logs:*"
      ],
      "Resource": "arn:aws:logs:*:*:*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "s3:GetObject"
      ],
      "Resource": "arn:aws:s3:::*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "sns:Publish"
      ],
      "Resource": "your sns topic ARN"
    }
  ]
}
```

3. Create an IAM role, the `executionrole`, and attach the policy you just created to the role.

   For instructions on creating the role, go to Creating a Role for an AWS Service (AWS Management Console) in *IAM User Guide*. Note the following:

   - In **Select Role Type**, click **AWS Service Roles**, and then select **AWS Lambda**.
   - In **Attach Policy**, select the policy you created in the previous step.

Next Step

Step 2.3: Create a Lambda Function (AWS CloudTrail Events) (p. 74)

Step 2.3: Create a Lambda Function (AWS CloudTrail Events)

Now you are ready to upload the deployment package to create Lambda function. You will also test invoke the function. At this point, you have not configured notification on your S3 bucket, so Amazon S3 will not invoke your function. But you can test your Lambda function by manually invoking it, which requires the following:

- Sample S3 event data identify bucket name and an object key.
- Sample CloudTrail log object in the S3 bucket.

  You want this sample log to include a record that has the `eventType` value set to "ec2.amazonaws.com" and the `eventName` value set to "CreateSecurityGroup". The Lambda function looks for event records with these values.

Follow the steps to test invoke your Lambda function.

1. Create a Lambda function.

   At the command prompt, run the following Lambda CLI `create-function` command using the *adminuser* `profile`. You will need to update the command by providing the .zip file path and the "executionrole" ARN.

```
$ aws lambda create-function \
--region us-west-2 \
--function-name CloudTrailEventProcessing  \
--zip-file fileb://file-path/CloudTrailEventProcessing.zip \
--role execution-role-arn \
--handler CloudTrailEventProcessing.handler \
--runtime nodejs \
--profile adminuser \
--timeout 10 \
--memory-size 1024
```

   Note that if you want you can upload the .zip file to an Amazon S3 bucket in the same AWS region, and then specify the bucket and object name in the preceding command. You will need to replace the `--zip-file` parameter by the `--code` parameter as shown:

```
--code S3Bucket=bucket-name,S3Key=zip-file-object-key
```

2. Note down the Lambda function ARN. You will need this ARN when you configure a bucket notification in the next section.

Next Step

Step 2.4: Invoke Your Lambda Function Manually (AWS CloudTrail Events) (p. 75)

Step 2.4: Invoke Your Lambda Function Manually (AWS CloudTrail Events)

In this section, you invoke your Lambda function manually using sample Amazon S3 event data that you will pass to the Lambda function. When Lambda function executes, it will read the S3 object (a sample CloudTrail log) from the bucket identified in the S3 event data, and publish an event to your SNS topic if the sample CloudTrail log reports use of a specific API. For this walkthrough, the API is the SNS API used to create a topic. That is, the CloudTrail log reports a record identifying "sns.amazonaws.com" as the `eventSource`, and "CreateTopic" as the `eventName`.

1. **Prepare to invoke the Lambda function manually**

   a. Save the following JSON (an example S3 event) in a file, `input.txt`.

   You will provide this sample event when you invoke your Lambda function. For more information about the S3 event structure, go to Event Message Structure in the *Amazon Simple Storage Service Developer Guide*.

```
{
    "Records":[
        {
            "eventVersion":"2.0",
            "eventSource":"aws:s3",
            "awsRegion":"us-west-2",
            "eventTime":"1970-01-01T00:00:00.000Z",
            "eventName":"ObjectCreated:Put",
            "userIdentity":{
                "principalId":"AIDAJDPLRKLG7UEXAMPLE"
            },
            "requestParameters":{
                "sourceIPAddress":"127.0.0.1"
            },
            "responseElements":{
                "x-amz-request-id":"C3D13FE58DE4C810",
              "x-amz-id-2":"FMyUVURIY8/IgAtTv8xRjskZQpcIZ9KG4V5Wp6S7S/JR
WeUWerMUE5JgHvANOjpD"
            },
            "s3":{
                "s3SchemaVersion":"1.0",
                "configurationId":"testConfigRule",
                "bucket":{
                    "name":"your bucket name",
                    "ownerIdentity":{
                        "principalId":"A3NL1KOZZKExample"
                    },
                    "arn":"arn:aws:s3:::mybucket"
                },
                "object":{
                    "key":"ExampleCloudTrailLog.json.gz",
                    "size":1024,
                    "eTag":"d41d8cd98f00b204e9800998ecf8427e",
                    "versionId":"096fKKXTRTtl3on89fVO.nfljtsv6qko"
                }
            }
        }
```

```
    ]
}
```

b. **Upload a sample CloudTrail log to your *examplebucket*.**

i. Save the following sample CloudTrail log to a file (`ExampleCloudTrailLog.json`).

Note that one of events in this log has "sns.amazonaws.com" as the `eventSource` and "CreateTopic" as the `eventName`. Your Lambda function reads the logs and if it finds event of this type, it publishes the event to the Amazon SNS topic you created and you will receive one email when you invoke the Lambda function manually.

```
{
   "Records":[
      {
         "eventVersion":"1.02",
         "userIdentity":{
            "type":"Root",
            "principalId":"account-id",
            "arn":"arn:aws:iam::account-id:root",
            "accountId":"account-id",
            "accessKeyId":"access-key-id",
            "sessionContext":{
               "attributes":{
                  "mfaAuthenticated":"false",
                  "creationDate":"2015-01-24T22:41:54Z"
               }
            }
         },
         "eventTime":"2015-01-24T23:26:50Z",
         "eventSource":"sns.amazonaws.com",
         "eventName":"CreateTopic",
         "awsRegion":"us-west-2",
         "sourceIPAddress":"205.251.233.176",
         "userAgent":"console.amazonaws.com",
         "requestParameters":{
            "name":"dropmeplease"
         },
         "responseElements":{
           "topicArn":"arn:aws:sns:us-west-2:account-id:exampletopic"

         },
         "requestID":"3fdb7834-9079-557e-8ef2-350abc03536b",
         "eventID":"17b46459-dada-4278-b8e2-5a4ca9ff1a9c",
         "eventType":"AwsApiCall",
         "recipientAccountId":"account-id"
      },
      {
         "eventVersion":"1.02",
         "userIdentity":{
            "type":"Root",
            "principalId":"account-id",
            "arn":"arn:aws:iam::account-id:root",
            "accountId":"account-id",
            "accessKeyId":"access-key-id",
            "sessionContext":{
               "attributes":{
```

```
                "mfaAuthenticated":"false",
                "creationDate":"2015-01-24T22:41:54Z"
             }
          }
       },
       "eventTime":"2015-01-24T23:27:02Z",
       "eventSource":"sns.amazonaws.com",
       "eventName":"GetTopicAttributes",
       "awsRegion":"us-west-2",
       "sourceIPAddress":"205.251.233.176",
       "userAgent":"console.amazonaws.com",
       "requestParameters":{
         "topicArn":"arn:aws:sns:us-west-2:account-id:exampletopic"

       },
       "responseElements":null,
       "requestID":"4a0388f7-a0af-5df9-9587-c5c98c29cbec",
       "eventID":"ec5bb073-8fa1-4d45-b03c-f07b9fc9ea18",
       "eventType":"AwsApiCall",
       "recipientAccountId":"account-id"
    }
  ]
}
```

ii. Run the `gzip` command to create .gz file from the preceding source file.

```
$ gzip ExampleCloudTrailLog.json
```

This creates `ExampleCloudTrailLog.json.gz` file.

iii. Upload the `ExampleCloudTrailLog.json.gz` file to your bucket that you specified in the CloudTrail configuration.

This object is specified in the sample Amazon S3 event data that we use in manually invoking the Lambda function.

2. Invoke the Lambda function.

Execute the following CLI command to invoke the function manually using *adminuser* `profile`.

```
$   aws lambda invoke-async \
 --function-name CloudTrailEventProcessing \
 --region us-west-2 \
 --invoke-args /filepath/input.txt \
 --debug \
--profile adminuser
```

Because your example log object has an event record showing the SNS API to call to create a topic, the Lambda function will post that event to your SNS topic, and you should get an email notification.

You can monitor the activity of your Lambda function by using CloudWatch metrics and logs. For more information about CloudWatch monitoring, see Troubleshooting and Monitoring AWS Lambda Functions with Amazon CloudWatch (p. 135).

Next Step

Step 3: Configure Amazon S3 to Publish Events (AWS CloudTrail Events) (p. 78)

Step 3: Configure Amazon S3 to Publish Events (AWS CloudTrail Events)

In this section, you add the remaining configuration so Amazon S3 can publish object-created events to AWS Lambda and invoke your Lambda function. You will do the following:

- Add permission to the Lambda function's access policy to allow Amazon S3 to invoke the function.
- Add notification configuration to your source bucket. In the notification configuration, you provide:
  - The event type for which you want Amazon S3 to publish events. For this exercise, you will specify the `s3:ObjectCreated:*` event type.
  - Lambda function to invoke.

Add permission to the Lambda Function's Access Policy

1. Run the following Lambda CLI `add-permission` command to grant Amazon S3 service principal ("s3.amazonaws.com") permission for the `lambda:InvokeFunction` action. Note that the permission is granted with the following conditions:

   - Amazon S3 can invoke the function only if an object-created event is detected on a specific bucket.
   - The bucket is owned by a specific AWS account. If a bucket owner deletes a bucket, some other AWS account can create a bucket with the same name. This condition ensures that only a specific AWS account can invoke your Lambda function.

   ```
   $ aws lambda add-permission \
   --function-name CloudTrailEventProcessing \
   --region us-west-2 \
   --statement-id Id-1 \
   --action "lambda:InvokeFunction" \
   --principal s3.amazonaws.com \
   --source-arn arn:aws:s3:::examplebucket \
   --source-account examplebucket-owner-account-id \
   --profile adminuser
   ```

2. Verify the access policy of your function by calling the CLI `get-policy` command.

   ```
   $ lambda get-policy \
   --function-name function-name \
   --profile adminuser
   ```

Step 3.2: Add Notification Configuration to Your Bucket

Add notification configuration on the *examplebucket* to request Amazon S3 to publish object-created events to Lambda. In the configuration, you specify the following:

- Event type — For this exercise, these can be any event types that create objects.
- Lambda function ARN – This is your Lambda function that Amazon S3 will invoke. The ARN is of the following form:

```
arn:aws:lambda:aws-region:account-id:function:function-name
```

For example, the function `CloudTrailEventProcessing` created in us-west-2 region will have this ARN:

```
arn:aws:lambda:us-west-2:account-id:function:CloudTrailEventProcessing
```

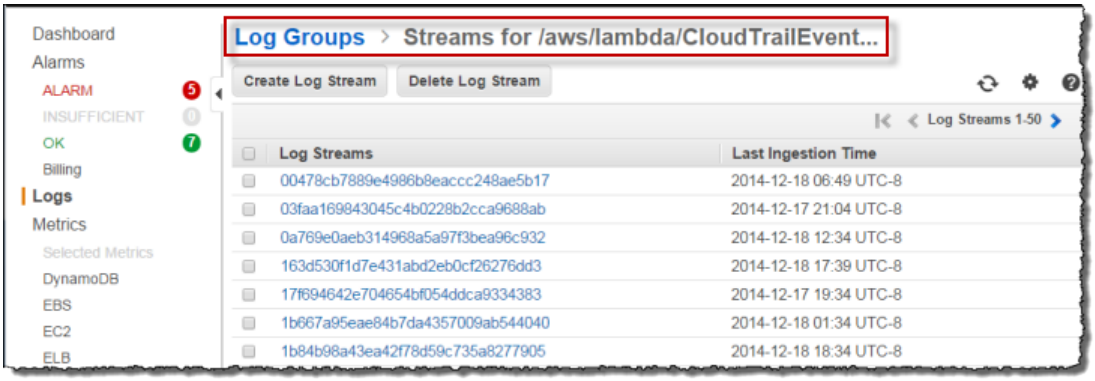
For instructions on adding notification configuration to a bucket, go to Enabling Event Notifications in the *Amazon Simple Storage Service Console User Guide*.

Step 3.3: Test the Setup

You are all done! You can now test the setup as follows:

1. Perform some action in your AWS account. For example, add another topic in the Amazon SNS console.
2. You should get an email notification about this event. You also notice the following:
   - AWS CloudTrail creates a log object in your bucket.
   - If you open the log object (.gz file), the log shows the `CreateTopic` SNS event.
   - For each object AWS CloudTrail creates, Amazon S3 invokes your Lambda function by passing in the log object as event data.
   - Lambda executes your function. The function parses the log, and it finds a `CreateTopic` SNS event, sends you an email notification.

     You can monitor the activity of your Lambda function by using CloudWatch metrics and logs. For more information about CloudWatch monitoring, see Troubleshooting and Monitoring AWS Lambda Functions with Amazon CloudWatch (p. 135).



AWS Lambda Walkthrough 6: Handling Mobile User Application Events for Android (Node.js)

Scenario

In this walkthrough, you will create a simple Android mobile application. The primary purpose of this walkthrough is to show you how to hook up various components to enable an Android mobile application to invoke a Lambda function and process response.

The mobile application processes events by invoking a Lambda function called ExampleAndroidEventProcessor using the AWS Mobile SDK for Android, and passing the event data to the function. The application events are not complex and the event data consists of a name (first name and last name) as shown:

```
{ firstName: 'value1', lastName: 'value2' }
```

The Lambda function then processes the event and sends back a response.

Use the following Node.js code to create the ExampleAndroidEventProcessor function in AWS Lambda.

```
exports.handler = function(event, context) {
   console.log("Received event: ", event);
  context.succeed("Hello "+ event.firstName + ". You are using " + context.cli
entContext.deviceManufacturer);
}
```

The code does the following two things only:

- The `console.log()` statement causes AWS Lambda to write logs Amazon CloudWatch Logs. In this case it writes the incoming event data to Amazon CloudWatch Logs.
- Upon successful execution, the `context.succeed()` method sets the response body to the string representation of its parameter. For information about the `context.succeed()` method , see Programming Model (Node.js) (p. 32). Your mobile code processes this response by simply displaying the message using the Android `Toast` class.

Note
The way that the mobile application invokes a Lambda function as shown in this walkthrough is an example of the AWS Lambda "request-response" model in which an application invokes a Lambda function and receives a response in real time. For more information, see Programming Model (Node.js) (p. 32).

Implementation Summary

Note the following about the mobile application:

- The mobile application must have valid security credentials and permissions to invoke a Lambda function.

  The mobile application in this walkthrough uses the Amazon Cognito service to manage user identities, authentication, and permissions. As part of the application setup, you create an Amazon Cognito identity pool to store user identities and define permissions. For more information, see Amazon Cognito.
- This example mobile application does not require a user to log in.

  A mobile application can require its users to log in using public identity providers such as Amazon and Facebook. The scope of this walkthrough is limited and assumes that the mobile application users are unauthenticated. It is essential that you understand this before you create an Amazon Cognito identity pool and configure user permissions.

For this walkthrough, configure the Amazon Cognito identity pool as follows:

- Enable access for unauthenticated identities.

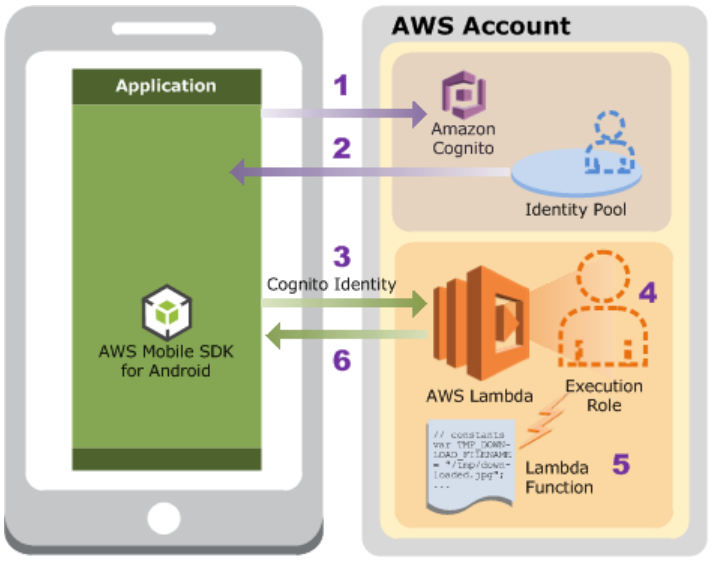
In this walkthrough, the mobile application users are unauthenticated (within the mobile application), so Amazon Cognito provides a unique identifier and temporary AWS credentials for these users to invoke the Lambda function.

- Add permission to invoke the Lambda function in the access policy associated with the IAM role for unauthenticated users.

  An identity pool has two associated IAM roles, one for authenticated and one for unauthenticated application users. Depending on the application user, Amazon Cognito assumes one of the roles when it generates temporary credentials for the user. In this example, Amazon Cognito assumes the role for unauthenticated users to obtain temporary credentials. Using the temporary credentials, the application user can then invoke the Lambda function.

  The access policy associated with the IAM role determines what the mobile application user can do when using the temporary credentials. In this walkthrough, you update the policy to allow permission to invoke the ExampleAndroidEventProcessor function.

The following diagram illustrates the application flow:



- Step 1: The mobile application sends the request to Amazon Cognito, and Amazon Cognito uses the identity pool ID in the requests.
- Step 2: The application receives temporary security credentials from Amazon Cognito.

  Amazon Cognito assumes the role associated with the identity pool and generates temporary credentials. What the application can do with the temporary credentials is limited by the access policy associated with the role. The AWS SDK can cache the temporary credentials so that the application does not send a request to Amazon Cognito each time it needs to invoke a Lambda function.
- Step 3: The mobile application invokes the Lambda function using temporary credentials (Cognito Identity).

AWS Lambda executes the function and responds immediately (in real time) with output as follows:

- Step 4: AWS Lambda assumes the execution role to execute your Lambda function on your behalf.
- Step 5: The Lambda function executes.
- Step 6: AWS Lambda returns results to the mobile application.

Now you are ready to try the steps.

Note that after the initial preparation, the walkthrough is divided into two main sections:

- First, you perform the necessary setup to create a Lambda function. Instead of creating an event source (the Android mobile application), you invoke the Lambda function manually using sample event data. This intermediate testing verifies that the function works.
- Second, you create an Amazon Cognito identity pool to manage authentication and permissions, and create the example Android application.

  When you run the Android application, it creates sample events and invokes your Lambda function.

Next Step

Step 1: Preparing for the Walkthrough (p. 82)

Step 1: Preparing for the Walkthrough

We recommend that you do not use the root credentials of your AWS account. Instead, create an administrator user in your account and use the administrator user credentials in setting up the walkthrough. If you already have an administrator user, you can skip this step.

For instructions to create an administrator user, see Set Up an AWS Account and Create an Administrator User (p. 12).

> **Note**
> The walkthrough assumes you are creating a Lambda function and an Amazon Cognito identity pool in the `us-west-2` region. If you want to use a different AWS region, make sure you create these resources in the same region. You also need to update the example mobile application code by providing the specific region that you want to use.

Next Step

Step 2: Create and Test the Lambda Function (p. 82)

Step 2: Create and Test the Lambda Function

Use the AWS Lambda console to create the Lambda function. Before you follow the steps, note the following:

- Specify ExampleAndroidEventProcessor as the function name.

  If you use any other name, you must specify that name in the access policy associated with the IAM role in the Amazon Cognito identity pool that you will create in the next section.
- Use the "Hello World" code template.
- Choose **Edit code inline** and replace the console-provided code by the following JavaScript code:

```
exports.handler = function(event, context) {
   console.log("Received event: ", event);
   context.succeed("Hello "+ event.firstName + "using " + context.clientCon
text.deviceManufacturer);
}
```

- For the execution role, select the **Basic Execution Role**.

Your JavaScript code (Lambda function) only writes logs to CloudWatch logs and the `console.log()` statement simply writes the incoming event to CloudWatch logs, so the permissions granted to the **Basic Execution Role** are sufficient for this application.

Now you are ready to create a Lambda function and test it manually. You can follow the steps in the Getting Started exercise to create and test the Lambda function.

- For instruction to create the Lambda function, see Step 1: Create a Lambda Function (p. 17).
- Test the function by manually invoking it in the Lambda console. Use the following sample event data in the console:

```
{
  "firstName": "first-name",
  "lastName": "last-name"
}
```

For instructions, see Step 2: Invoke the Lambda Function Manually (p. 19).

Next Step

Step 3: Create an Amazon Cognito Identity Pool (p. 83)

Step 3: Create an Amazon Cognito Identity Pool

In this section, you create an Amazon Cognito identity pool. The identity pool has two IAM roles. You update the IAM role for unauthenticated users and grant permission to execute the ExampleAndroidEventProcessor Lambda function.

For more information about IAM roles, go to IAM Roles (Delegation and Federation) in the *IAM User Guide*.

For more information about Amazon Cognito services, go to the Amazon Cognito product detail page.

To create an identity pool

1. Using the IAM User Sign-In URL, sign in to the Amazon Cognito console as *adminuser*.
2. Create a new identity pool called ExampleAndroidEventProcessorPool. Before you follow the procedure to create an identity pool, note the following:

   - The identity pool you are creating must allow access to unauthenticated identities because our example mobile application does not require a user log in (the application users are unauthenticated). In this case, select the **`Enable access to unauthenticated identities`** option.
   - The unauthenticated application users need permission to invoke the Lambda function. To enable this, add the following statement to allow the `lambda:InvokeFunction` action for the specific Lambda function.

```
{
        "Effect": "Allow",
        "Action": [
            "lambda:InvokeFunction"
        ],
        "Resource": [
           "arn:aws:lambda:us-west-2:account-id:function:ExampleAndroid
```

```
EventProcessor"
        ]
}
```

The `Resource` ARN identifies your Lambda function for which you are granting the `lambda:InvokeFunction` action and ExampleAndroidEventProcessor is the Lambda function name that you created in the preceding step.

Update the access policy when the identity pool is being created. The resulting policy will be as follows:

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Effect":"Allow",
         "Action":[
            "mobileanalytics:PutEvents",
            "cognito-sync:*"
         ],
         "Resource":[
            "*"
         ]
      },
      {
         "Effect":"Allow",
         "Action":[
            "lambda:invokefunction"
         ],
         "Resource":[
            "arn:aws:lambda:us-west-2:account-id:function:ExampleAndroid
EventProcessor"
         ]
      }
   ]
}
```

Note
After creating the identity pool, go to the IAM console, find the role for unauthenticated users, and edit the access policy. Make sure you write down the IAM role name for the unauthenticated users so that you can search for it in the IAM console.

For instructions about how to create an identity pool, log in to the Amazon Cognito console and follow the **New Identity Pool** wizard.

Write down the identify pool ID. You specify this ID in your mobile application. The application uses this ID when it sends request to Amazon Cognito to request for temporary security credentials.

Next Step

Step 4: Create a Mobile User Application for Android (p. 85)

Step 4: Create a Mobile User Application for Android

Now you can create a simple Android mobile application that generates events and invokes Lambda functions by passing the event data as parameters.

The following instructions have been verified using Android studio.

1. Create a new Android project called AndroidEventGenerator using the following configuration:
   - Select the **Phone and Tablet** platform.
   - Choose **Blank Activity**.
2. In the build.gradle (Module:app) file, add the following in the `dependencies` section:

```
compile 'com.amazonaws:aws-android-sdk-core:2.2.+'
compile 'com.amazonaws:aws-android-sdk-lambda:2.2.+'
```

3. Build the project so that the required dependencies are downloaded, as needed.
4. In the Android application manifest (AndroidManifest.xml), add the following permissions so that your application can connect to the Internet. You can add them just before the `</manifest>` end tag.

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

5. In MainActivity, add the following imports:

```
import com.amazonaws.mobileconnectors.lambdainvoker.*;
import com.amazonaws.auth.CognitoCachingCredentialsProvider;
import com.amazonaws.regions.Regions;
```

6. In the `package` section, specify the appropriate package name (as shown below *`com.example.....lambdaeventgenerator`*). Add a new class called `NameInfo`, instances of which act as the POJO (Plain Old Java Object) for event data which consists of first and last name.

```
package com.example....lambdaeventgenerator;
public class NameInfo {
    private String firstName;
    private String lastName;

    public NameInfo() {}

    public NameInfo(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
```

```
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

7. In the specified package (as shown below *com.example.....lambdaeventgenerator*), create an interface called MyInterface for invoking the ExampleAndroidEventProcessor Lambda function. Note that the `@LambdaFunction` annotation in the code maps the specific client method to the same-name Lambda function. For more information about this annotation, go to AWS Lambda in the *AWS SDK for Android Developer Guide*.

```
package com.example.....lambdaeventgenerator;
import com.amazonaws.mobileconnectors.lambdainvoker.LambdaFunction;
public interface MyInterface {

    /**
     * Invoke the Lambda function "ExampleAndroidEventProcessor".
     * The function name is the method name.
     */
    @LambdaFunction
    String ExampleAndroidEventProcessor(NameInfo nameInfo);

}
```

8. To keep the application simple, we are going to add code to invoke the Lambda function in the `onCreate()` event handler. In MainActivity, add the following code toward the end of the `onCreate()` code.

```

// Create an instance of CognitoCachingCredentialsProvider
CognitoCachingCredentialsProvider cognitoProvider = new CognitoCachingCreden
tialsProvider(
       this.getApplicationContext(), "Identity-pool-id", Regions.US_WEST_2);

// Create LambdaInvokerFactory, to be used to instantiate the Lambda proxy.
LambdaInvokerFactory factory = new LambdaInvokerFactory(this.getApplication
Context(),
               Regions.UW_WEST_2, cognitoProvider);

// Create the Lambda proxy object with a default Json data binder.
// You can provide your own data binder by implementing
// LambdaDataBinder.
final MyInterface myInterface = factory.build(MyInterface.class);

NameInfo nameInfo = new NameInfo("John", "Doe");
// The Lambda function invocation results in a network call.
// Make sure it is not called from the main thread.
new AsyncTask<NameInfo, Void, String>() {
    @Override
    protected String doInBackground(NameInfo... params) {
        // invoke "echo" method. In case it fails, it will throw a
        // LambdaFunctionException.
```

```
        try {
            return myInterface.ExampleAndroidEventProcessor(params[0]);
        } catch (LambdaFunctionException lfe) {
            Log.e("Tag", "Failed to invoke echo", lfe);
            return null;
        }
    }

    @Override
    protected void onPostExecute(String result) {
        if (result == null) {
            return;
        }

        // Do a toast
        Toast.makeText(MainActivity.this, result, Toast.LENGTH_LONG).show();

    }
}.execute(nameInfo);
```

9. Run the code and verify as follows:
   - The `Toast.makeText()` displays the response returned.
   - Verify that CloudWatch Logs shows the log created by the Lambda function. It should show the event data (first name and last name). You can also verify this in the AWS Lambda console.

Authoring Lambda Functions in Java

This section explains how to author your Lambda functions in Java. We recommend you first review the information in AWS Lambda: How it Works (p. 3) and make sure you are familiar with core AWS Lambda concepts such as function, event source, event source mapping, Lambda permission model, and resource model. You can then review the topics in this section for information specific to creating Lambda functions in Java.

Topics

Getting Started: Authoring AWS Lambda Code in Java

Topics

Introduction

In this Getting Started exercise, you will use the following Java code example to create your Lambda function.

```
package example;
```

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;

public class Hello {
    public String myHandler(int myCount, Context context) {
        LambdaLogger logger = context.getLogger();
        logger.log("received : " + myCount);
        return String.valueOf(myCount);
    }
}
```

The programming model explains how to write your Java code in detail, for example the input/output types AWS Lambda supports. For more information about the programming model, see Programming Model for Authoring Lambda Functions in Java (p. 101). For now, note the following about this code:

- When you package and upload this code to create your Lambda function, you will specify the `example.Hello::myHandler` method reference as the handler.

  The name "myHandler" is arbitrary. You can name the method anything you want.

  When AWS Lambda executes the Lambda function, it invokes this handler. The first parameter is the input to the handler which can be event data (published by an event source) or custom input you provide such as a string or any custom data object. In order for AWS Lambda to successfully invoke this handler the function must be invoked with input data that can be serialized into the data type of the input parameter.
- The handler in this example uses int type for input and String for output.

  AWS Lambda supports input/output of primitive Java types (such as String and int), POJO types, and Stream types. For this example, the handler uses int type for input and String type for output - when you invoke this function you will pass a sample int (for example, 123).
- The code specifies the `return` statement.

  The output value of the code (specified in the return statement) is used differently depending on how the function is invoked:

  - `RequestResponse` invocation type: In this case Lambda function can return response in real-time. This is used for synchronous applications as well as testing your function using the AWS Lambda console. In this case, the response gets returned to the caller; for example, if you invoke the above sample using the AWS Lambda console, you will see a string output as a result.

    Note
    We recommend not using void response so you can test response in real-time because the console uses the `RequestResponse` as the invocation type.
  - `Event` invocation type: In this case AWS Lambda executes the function asynchronously. When you use AWS Lambda with event sources such as Amazon S3, Amazon Kinesis, and Amazon SNS, these event source invoke the Lambda function using the `Event` invocation type. In this case, the response type is not persisted or used anywhere.

  You will test the Lambda function using the console. And console supports only the "RequestResponse" invocation type. Any results the Lambda function returns will appear in the console.
- The handler includes the optional `Context` parameter. In the code we use the `LambdaLogger` provided by the `Context` object to write log entries to CloudWatch logs. For information about using the Context object, see The Context Object (Java) (p. 112).

First, you need to package this code and any dependencies into a deployment package. Then, you upload the deployment package to AWS Lambda to create your Lambda function. Lastly, you test the code by invoking the Lambda function manually using sample event data.

Step 1: Create Deployment Package

Your deployment package can be a .zip file or a standalone .jar. You can use any build and packaging tool you are familiar with to create a deployment package. The following sections provide examples of how to use Maven build tool to create a standalone .jar and how to use a Gradle build tool to create a .zip file:

- If you want to create a .jar file as your deployment package, click one of the following links:

  Creating a .jar Deployment Package Using Maven without any IDE (Java) (p. 93)

  Creating a .jar Deployment Package Using Maven and Eclipse IDE (Java) (p. 96)
- If you want to create a .zip file as your deployment package, click the following link:

  Creating a .zip Deployment Package (Java) (p. 98)

After you verify your deployment package is created, go the next section to upload the package to AWS Lambda to create a Lambda function.

Step 2: Create Lambda Function

This section provides steps to create a Lambda function using the AWS Management Console and the AWS CLI.

> **Note**
> For this Getting Started exercise, we assume you are creating the Lambda function in the US West (Oregon) region.

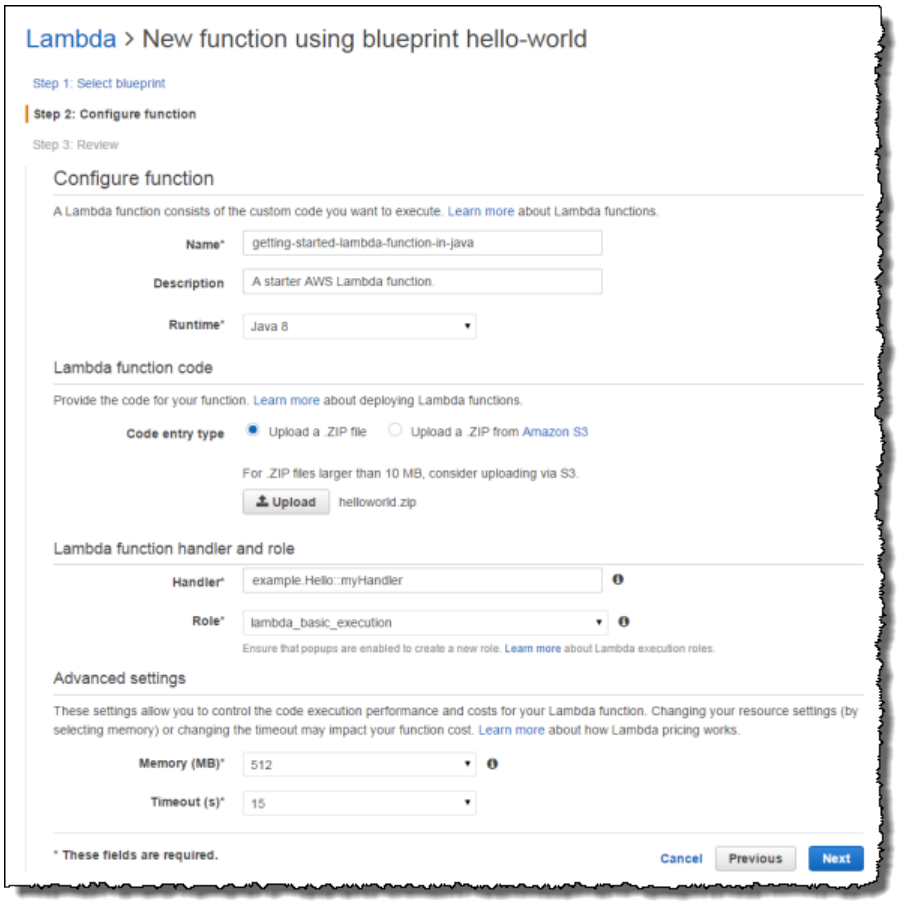
To create a Lambda function using AWS Management Console

1. Sign in to the AWS Management Console, open the AWS Lambda console and make sure that you have the US West (Oregon) region selected.
2. Choose **Create a Lambda function**.
3. In **Step 1: Select blueprint**, choose the **hello-world** blueprint.
4. In **Step 2: Configure function**, specify the following values.

   - Enter a function name (for example, **`getting-started-lambda-function-in-java`**).
   - Select **Java 8** from the **Runtime** list.
   - Choose **Upload a .ZIP file**, click **Upload**, and then choose the .jar (or .zip file) you created in the preceding section.

     Note that you can also upload the .jar or .zip file to an S3 bucket, and provide the S3 bucket name and object key.
   - In **Handler**, specify *package*.*class-name*::*handler* (in the Java code in this example, `example.Hello::myHandler`).
   - For the IAM **Role**, create a role by choosing the **Basic execution role** from the list.

     If this is your first time, AWS Management Console will create an IAM role called `basic_execution_role` in your account with an access policy that allows only permission to write logs to CloudWatch Logs. Our example Java code does not access any other AWS resources so these permissions are sufficient.
   - In **Memory**, specify 512.
   - In **Timeout**, specify 15 seconds.



5. In **Step 3: Review**, click **Create Function**.

To create Lambda function using AWS CLI

1. At the command prompt, make sure you are in the project directory (*`project-dir`*).
2. In the IAM console create an execution role called **lambda_basic_execution** with the following access policy:

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

Note
If you previously used the Lambda console and specified the basic execution role as the role, the console already has created the lambda_basic_execution role with this access

policy. You can skip the step to create the role, but you will need to get the role ARN from the IAM console.

3. Write down the lambda_basic_execution role ARN. You will need the role ARN when creating your Lambda function.
4. Run the following `create-function` command to create a `getting-started-lambda-function-in-java` Lambda function.

   You need to update the command by providing the execution role. Note the command specifies `java8` as the runtime because your Lambda function code is written in Java.

```
aws lambda create-function \
--region us-west-2 \
--function-name getting-started-lambda-function-in-java \
--zip-file fileb://deployment-package (zip or jar) path \
--role arn:aws:iam::account-id:role/lambda_basic_execution  \
--handler example.Hello::myHandler \
--runtime java8 \
--timeout 15 \
--memory-size 512
```

   Note the `--handler` parameter identifies *package.class*::*handler* in your Java code. When AWS Lambda executes this Lambda function it will invoke the handler you specify when creating the Lambda function.

   AWS Lambda creates a Lambda function and returns a response. An example is shown:

```
  {
        "FunctionName": "getting-started-lambda-function-in-java",
        "CodeSize": 22617,
        "MemorySize": 512,
       "FunctionArn": "arn:aws:lambda:us-west-2:account-id:function:getting-
started-lambda-function-in-java",
        "Handler": "example.Hello::handler",
        "Role": "arn:aws:iam::account-id:role/lambda_basic_execution",
        "Timeout": 15,
        "LastModified": "2015-05-30T23:15:26.716+0000",
        "Runtime": "java8",
        "Description": ""
}
```

Step 3: Test the Lambda Function

Now you have a Lambda function created in AWS Lambda. The Lambda function handler can receive input data as int and returns the string representation of the same. The function also logs a string "received: *input int*" to CloudWatch logs.

For this walkthrough, you don't write an application that generates int inputs and invokes your Lambda function. Instead, you manually invoke the Lambda function using a sample int as the input data. You can invoke the Lambda function manually using either the console or the AWS CLI.

Manually invoke a Lambda function using the console

1. Sign in to the AWS Management Console and open the AWS Lambda console, and make sure you have the US West (Oregon) region selected.

2. Choose the function.
3. In the function page, click **Test**.
4. The **Input sample event** page will show Hello World sample event. Replace the sample event by an integer as sample event data (for example, 123) and click **Submit**.

   If you review the Lambda function, you will notice that the first parameter of the handler (that is, `myHandler`) is of int type, so you pass a int input.

   The console sends an `invoke` request using `RequestResponse` invocation type (that is, synchronous execution). AWS Lambda executes the function on your behalf by invoking the handler. AWS Lambda passes the event data (in this case, an integer say 123) to the handler as the first parameter. The Lambda function executes and returns a string (in this case, "123") back in real-time.

Manually invoke a Lambda function using AWS CLI

1. Run the following CLI command to invoke your Lambda function.

```
aws lambda invoke \
--region us-west-2 \
--function-name getting-started-lambda-function-in-java \
--payload 123 \
--invocation-type RequestResponse \
/tmp/response
```

   Note that `RequestResponse` is the invocation type by default. The parameter is specified only for readability.
2. After AWS Lambda executes the function, it returns results in real-time. You can verify the output in the `/tmp/response` file.

Creating a Deployment Package (Java)

Your deployment package can be a .zip file or a standalone jar, it is your choice. You can use any build and packaging tool you are familiar with to create a deployment package.

We provide examples of using Maven to create standalone jars and using Gradle to create a .zip file. For more information, see the following topics:

Topics

Creating a .jar Deployment Package Using Maven without any IDE (Java)

This section shows how to package your Java code into a deployment package using Maven at the command line.

Topics

Before You Begin

You will need to install the Maven command-line build tool. For more information, go to Maven. If you are using Linux, check your package manager.

```
sudo apt-get install mvn
```

if you are using Homebrew

```
brew install maven
```

Project Structure Overview

After you set up the project, you should have the following folder structure:

```
proj-dir/pom.xml
proj-dir/src/main/java  (your code goes here)
```

Your code will then be in the /java folder. For example, if your package name is "example" and you have a Hello.java class in it, the structure will be:

```
proj-dir/src/main/java/example/Hello.java
```

After you build the project, the resulting .jar file (that is, your deployment package), will be in the *proj-dir*`/target` subdirectory.

Step 1: Create Project

Follow the steps in this section to create a Java project.

1. Create a project directory (*project-dir*).
2. In the *project-dir* directory, create the following:

   - Project Object Model file, `pom.xml` file. Add the following project information and configuration details for Maven to build the project.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="ht
tp://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 ht
tp://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>doc-examples</groupId>
  <artifactId>lambda-java-example</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>lambda-java-example</name>
```

```
  <dependencies>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-core</artifactId>
      <version>1.0.0</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>
```

Note

- In the `dependencies` section, the `groupId` (that is, com.amazonaws) is the Amazon AWS group ID for Maven artifacts in the Maven Central Repository. The `artifactId` (that is, aws-lambda-java-core) is the AWS Lambda core library that provides definitions of the `RequestHandler`, `RequestStreamHandler`, and the `Context` AWS Lambda interfaces for use in your Java application. At the build time Maven resolves these dependencies.
- In the plugins section, the Apache `maven-shade-plugin` is a plugin that Maven will download and use during your build process. This plugin is used for packaging jars to create a standalone .jar (a .zip file), your deployment package.
- If you are following other walkthrough topics in this guide, the specific walkthroughs might require you to add more dependencies. Make sure to add those dependencies as required.

3. In the *`project-dir`*, create the following structure:

```
project-dir/src/main/java
```

4. Under the `/java` subdirectory you add your Java files and folder structure, if any. For example, if you Java package name is "example", and source code is "Hello.java", your directory structure looks like this:

```
project-dir/src/main/java/example/Hello.java
```

Step 2: Build Project (Create Deployment Package)

Now you can build the project using Maven at the command line.

1. At a command prompt change directory to the project directory (*project-dir*).
2. Run the following `mvn` command to build the project:

```
$ mvn package
```

The resulting .jar is saved as *project-dir*`/target/lambda-java-example-1.0-SNAPSHOT.jar`. The .jar name is created by concatenating the `artifactId` and `version` in the POM.xml file.

The build creates this resulting .jar, using information in the pom.xml to do the necessary transforms. This is a standalone .jar (.zip file) that includes all the dependencies. This is your deployment package that you can upload to AWS Lambda to create a Lambda function.

Creating a .jar Deployment Package Using Maven and Eclipse IDE (Java)

This section shows how to package your Java code into a deployment package using Eclipse IDE and Maven plugin for Eclipse.

Topics

Before You Begin

Install the **Maven** Plugin for Eclipse.

1. Start Eclipse. From the **Help** menu in Eclipse, choose **Install New Software**.
2. In the **Install** window, type **`http://download.eclipse.org/technology/m2e/releases`** in the **Work with:** box, and choose **Add**.
3. Follow the steps to complete the setup.

Step 1: Create and Build a Project

In this step, you start Eclipse and create a Maven project. You will add the necessary dependencies, and build the project. The build will produce a .jar, which is your deployment package.

1. Create a new Maven project in Eclipse.
   a. From the **File** menu, choose **New**, and then choose **Project**.
   b. In the **New Project** window, choose **Maven Project**.
   c. In the **New Maven Project** window, choose **Create a simple project**, and leave other default selections.

d. In the **New Maven Project**, **Configure project** windows, type the following **Artifact** information:

- **Group Id**: doc-examples
- **Artifact Id**: lambda-java-example
- **Version**: 0.0.1-SNAPSHOT
- **Packaging**: jar
- **Name**: lambda-java-example

2. Add the `aws-lambda-java-core` dependency to the pom.xml file.

   It provides definitions of the `RequestHandler`, `RequestStreamHandler`, and `Context` interfaces. This allows you to compile code that you can use with AWS Lambda.

   a. Open the context (right-click) menu for the pom.xml file, choose **Maven**, and then choose **Add Dependency**.
   b. In the **Add Dependency** windows, type the following values:

      Group Id: com.amazonaws

      Artifact Id: aws-lambda-java-core

      Version: 1.0.0

      Caution
      If you are following other walkthrough topics in this guide, the specific walkthroughs might require you to add more dependencies. Make sure to add those dependencies as required.

3. Add Java class to the project.

   a. Open the context (right-click) menu for the `src/main/java` subdirectory in the project, choose **New**, and then choose **Class**.
   b. In the **New Java Class** window, type the following values:

      - **Package**: example
      - **Name**: Hello

      Caution
      If you are following other walkthrough topics in this guide, the specific walkthroughs might recommend different package name or class name.

   c. Add your Java code. If you are following other walkthrough topics in this guide, add the provided code.

4. Build the project.

   Open the context (right-click) menu for the project in **Package Explorer**, choose **Run As**, and then choose **Maven Build**. In the **Edit Configuration** window, type "package" in the **Goals** box.

   Note
   The resulting .jar, **lambda-java-example-0.0.1-SNAPSHOT.jar**, is not the final standalone .jar that you can use as your deployment package. In the next step, you add the Apache

`maven-shade-plugin` to create the standalone .jar. For more information, go to Apache Maven Shade Plugin.

5. Add the `maven-shade-plugin` plugin and rebuild.

   The maven-shade-plugin will take artifacts (jars) produced by the *package* goal (produces customer code .jar), and created a standalone .jar that contains the compiled customer code, and the resolved dependencies from the pom.xml.

   a. Open the context (right-click) menu for the pom.xml file, choose **Maven**, and then choose **Add Plugin**.
   b. In the **Add Plugin** window, type the following values:

      - **Group Id:** org.apache.maven.plugins
      - **Artifact Id:** maven-shade-plugin
      - **Version:** 2.3

   c. Now build again.

      This time we will create the jar as before, and then use the `maven-shade-plugin` to pull in dependencies to make the standalone .jar.

      i. Open the context (right-click) menu for the project, choose **Run As**, and then choose **Maven build**.
      ii. In the **Edit Configuration** windows, type `package shade:shade` in the **Goals** box.
      iii. Choose `Run`.

      You can find the resulting standalone .jar (that is, your deployment package), in the `/target` subdirectory.

      Open the context (right-click) menu for the `/target` subdirectory, choose **Show In**, choose **System Explorer**, and you will find the `lambda-java-example-0.0.1-SNAPSHOT.jar`.

Creating a .zip Deployment Package (Java)

This section provides examples of creating .zip file as your deployment package. You can use any build and packaging tool you like to create this zip. Regardless of the tools you use, the resulting .zip file must have the following structure:

- All compiled class files and resource files at the root level.
- All required jars to run the code in the `/lib` directory.

Note
You can also build a standalone .jar (also a zipped file) as your deployment package. For examples of creating standalone .jar using Maven, see Creating a Deployment Package (Java) (p. 93).

The following examples use Gradle build and deployment tool to create the .zip.

Important
Minimum Gradle version 2.0 is required.

Before You Begin

You will need to download Gradle. For instructions, go to the gradle website, https://gradle.org/ .

Example 1: Creating .zip Using Gradle and the Maven Central Repository

At the end of this walkthrough, you will have a project directory (*proj-dir*) with content having the following structure:

```
proj-dir/build.gradle
proj-dir/src/main/java
```

The `/java` folder will contain your code. For example, if your package name is "example", and you have a Hello.java class in it, the structure will be:

```
proj-dir/src/main/java/example/Hello.java
```

After you build the project, the resulting .zip file (that is, your deployment package), will be in the *proj-dir*`/build/distributions` subdirectory.

1. Create a project directory (*proj-dir*).
2. In the *proj-dir* create build.gradle file and add the following content:

```
apply plugin: 'java'

repositories {
    mavenCentral()
}

dependencies {
    compile (
        'com.amazonaws:aws-lambda-java-core:1.0.0',
        'com.amazonaws:aws-lambda-java-events:1.0.0'
    )
}

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

build.dependsOn buildZip
```

Note

- The repositories section refers to Maven Central Repository. At the build time, it fetches the dependencies (that is, the two AWS Lambda libraries) from Maven Central.
- The `buildZip` task describes how to create the deployment package .zip file.

For example, if you unzip the resulting .zip file you should find any of the compiled class files and resource files at the root level. You should also find a `/lib` directory with the required jars for running the code.

- If you are following other walkthrough topics in this guide, the specific walkthroughs might require you to add more dependencies. Make sure to add those dependencies as required.

3. In the *`project-dir`*, create the following structure:

```
project-dir/src/main/java
```

4. Under the `/java` subdirectory you add your Java files and folder structure, if any. For example, if you Java package name is "example", and source code is "Hello.java", then your directory structure looks like this:

```
project-dir/src/main/java/example/Hello.java
```

5. Run the following gradle command to build and package the project in a .zip file.

```
proj-dir> gradle build
```

6. Verify the resulting *`proj-dir`*`.zip` file in the *`proj-dir`*`\build\distributions` subdirectory.
7. Now you can upload the .zip file, your deployment package to AWS Lambda to create a Lambda function and test it by manually invoking it using sample event data. For instruction, see Getting Started: Authoring AWS Lambda Code in Java (p. 88).

Example 2: Creating .zip Using Gradle Using Local Jars

You may choose not to use the Maven Central repository. Instead have all the dependencies in the project folder. In this case your project folder (proj-dir) will have the following structure:

```
proj-dir/jars           (all jars go here)
proj-dir/build.gradle
proj-dir/src/main/java  (your code goes here)
```

So if your Java code has `example` package and `Hello.java` class, the code will be in the following subdirectory:

```
proj-dir/src/main/java/example/Hello.java
```

You build.gradle file should be as follows:

```
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
```

Note that the dependencies specify `fileTree` which identifies *`proj-dir`*`/jars` as the subdirectory that will include all the required jars.

Now you build the package. Run the following gradle command to build and package the project in a .zip file.

```
proj-dir> gradle build
```

Authoring Lambda Functions Using Eclipse IDE and AWS SDK Plugin (Java)

Topics

AWS SDK Eclipse Toolkit provides an Eclipse plugin for you to both create a deployment package and also upload it to create a Lambda function. If you can use Eclipse IDE as your development environment, this plugin enables you to author Java code, create and upload a deployment package and create your Lambda function. For more information, go to AWS Toolkit for Eclipse Getting Started Guide.

Programming Model for Authoring Lambda Functions in Java

Your Lambda function code must be written in a stateless style, and have no affinity with the underlying compute infrastructure. Your code should expect local file system access, child processes, and similar artifacts to be limited to the lifetime of the request, and store any persistent state in Amazon S3, Amazon DynamoDB, or another cloud storage service. Requiring functions to be stateless enables AWS Lambda to launch as many copies of a function as needed to scale to the incoming rate of events and requests. These functions may not always run on the same compute instance from request to request, and a given instance of your Lambda function may be used more than once by AWS Lambda.

AWS Lambda provides the following two libraries:

- `aws-lambda-java-core`: This library provides the Context object, RequestStreamHandler, and the RequestHandler interfaces. The `Context` object (The Context Object (Java) (p. 112)) provides runtime information about your Lambda function. The predefined interfaces provide one way of defining your Lambda function handler. For more information, see Leveraging Predefined Interfaces for Creating Handler (Java) (p. 108).
- `aws-lambda-java-events`: This library provides predefined types that you can use when writing Lambda functions to process events published by Amazon S3, Amazon Kinesis, Amazon SNS, and Amazon Cognito. These classes help you process the event without having to write your own custom serialization logic.

These libraries are not required but are provided as convenience for you when writing your Lambda code. These libraries are available through the Maven Central Repository and can also be found on GitHub.

Topics

Lambda Function Handler (Java)

At the time you create a Lambda function you specify a handler that AWS Lambda can invoke when the service executes the Lambda function on your behalf.

Lambda supports two approaches for creating a handler:

- Loading handler method directly without having to implement an interface. This section describes this approach.
- Implementing standard interfaces provided as part of `aws-lambda-java-core` library (interface approach). For more information, see Leveraging Predefined Interfaces for Creating Handler (Java) (p. 108).

The general syntax for the handler is as follows:

```
outputType handler-name(inputType input, Context context) {
   ...
}
```

In order for AWS Lambda to successfully invoke a handler it must be invoked with input data that can be serialized into the data type of the `input` parameter.

In the syntax, note the following:

- *`inputType`*: The first handler parameter is the input to the handler, which can be event data (published by an event source) or custom input that you provide such as a string or any custom data object. In order for AWS Lambda to successfully invoke this handler, the function must be invoked with input data that can be serialized into the data type of the `input` parameter.
- *`outputType`*: If you plan to invoke the Lambda function synchronously (using the `RequestResponse` invocation type), you can return the output of your function using any of the supported data types. For example, if you use a Lambda function as a mobile application backend, you are invoking it synchronously. Your output data type will be serialized into JSON.

  If you plan to invoke the Lambda function asynchronously (using the `Event` invocation type), the `outputType` should be `void`. For example, if you use AWS Lambda with event sources such as Amazon S3, Amazon Kinesis, and Amazon SNS, these event sources invoke the Lambda function using the `Event` invocation type.
- The *`inputType`* and *`outputType`* can be one of the following:
  - Primitive Java types (such as String or int).
  - Predefined AWS event types defined in the `aws-lambda-java-events` library.

    For example `S3Event` is one of the POJOs predefined in the library that provides methods for you to easily read information from the incoming Amazon S3 event.

  - You can also write your own POJO class. AWS Lambda will automatically serialize and deserialize input and output JSON based on the POJO type.

  For more information, see Handler Input/Output Types (Java) (p. 104).
- You can omit the `Context` object from the handler method signature if it isn't needed. For more information, see The Context Object (Java) (p. 112).

For example, consider the following Java example code used in the Getting Started exercise for Java. For more information, see Getting Started: Authoring AWS Lambda Code in Java (p. 88).

```
package example;

import com.amazonaws.services.lambda.runtime.Context;

public class Hello {
    public String myHandler(int myCount, Context context) {
        return myCount.toString();
    }
}
```

In this example input is int type and output is String type. You packaged this example code and dependencies created a Lambda function, and specified `example.Hello::myHandler` (*`package`*.*`class`*::*`method-reference`*) as the handler, as shown in the following `create-function` CLI command from the Getting Started exercise for Java.

```
aws lambda create-function \
--region us-west-2 \
--function-name getting-started-lambda-function-in-java \
--zip-file fileb://deployment-package (zip or jar) path \
--role arn:aws:iam::account-id:role/lambda_basic_execution  \
--handler example.Hello::myHandler \
--runtime java8 \
--timeout 15 \
--memory-size 512
```

In the example Java code, the first handler parameter is the input to the handler (myHandler), which can be event data (published by an event source such as Amazon S3) or custom input you provide such as a int (as in this example) or any custom data object.

Handler Overload Resolution

If your Java code contains multiple methods with same name as the `handler` name, then AWS Lambda uses the following rules to pick a method to invoke:

1. Select the method with the largest number of parameters.
2. If two or more methods have the same number of parameters, then AWS Lambda selects the method that has the `Context` as the last parameter.

   If none or all of these methods have the `Context` parameter, then the behavior is undefined.

Additional Information

The following topics provide more information about the handler.

- For more information about the handler input and output types, see Handler Input/Output Types (Java) (p. 104).
- For information about using predefined interfaces to create a handler, see Leveraging Predefined Interfaces for Creating Handler (Java) (p. 108).

  If you implement these interfaces, they can help you compile time validation for your handler method signature.
- If an exception comes out of your Lambda function, AWS Lambda records metrics that an error occurred in CloudWatch. For more information, see Exceptions (Java) (p. 115).

Handler Input/Output Types (Java)

When AWS Lambda executes the Lambda function, it invokes this handler. The first parameter is the input to the handler which can be event data (published by an event source) or custom input you provide such as a string or any custom data object.

AWS Lambda supports the following input/output types for a handler:

- Simple Java types (AWS Lambda supports the String, Integer, Boolean, Map, and List types)
- POJO (Plain Old Java Object) type
- Stream type (If you do not want to use POJOs or if Lambda's serialization approach does not meet your needs, you can use the byte stream implementation. For more information, see Example: Using Stream for Handler Input/Output (Java) (p. 107).)

Handler Input/Output: String Type

The following Java class shows a handler called `myHandler` that uses String type for input and output.

```
package example;

import com.amazonaws.services.lambda.runtime.Context;

public class Hello {
    public String myHandler(String name, Context context) {
        return String.format("Hello %s.", name);
    }
}
```

You can have similar handler functions for other simple Java types.

> **Note**
> The return type should be void if you plan to invoke the Lambda function asynchronously (using `Event` invocation type). For more information, see Invoke (p. 178).

To test an end-to-end example, see Getting Started: Authoring AWS Lambda Code in Java (p. 88).

Handler Input/Output: POJO Type

The following Java class shows a handler called `myHandler` that uses POJOs for input and output.

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
```

```
public class HelloPojo {

    // Define two classes/POJOs for use with Lambda function.
    public static class RequestClass {
      ...
    }

    public static class ResponseClass {
      ...
    }

   public static ResponseClass myHandler(RequestClass request, Context context){

       String greetingString = String.format("Hello %s, %s.", request.getFirst
Name(), request.getLastName());
        return new ResponseClass(greetingString);
    }
}
```

AWS Lambda serializes based on standard bean naming conventions (see The Java EE 6 Tutorial). You should use mutable POJOs with public getters and setters.

> **Note**
> You shouldn't rely on any other features of serialization frameworks such as annotations. If you need to customize the serialization behavior, you can use the raw byte stream to use your own serialization.

If you use POJOs for input and output, you need to provide implementation of the `RequestClass` and `ResponseClass` types. For an example, see Example: Using POJOs for Handler Input/Output (Java) (p. 105).

Example: Using POJOs for Handler Input/Output (Java)

Suppose your application events generate data that includes first name and last name as shown:

```
{ firstName: 'John', lastName: 'Doe' }
```

For this example, the handler receives this JSON and returns the string "Hello John Doe".

```
public static ResponseClass myHandler(RequestClass request, Context context){
       String greetingString = String.format("Hello %s, %s.", request.firstName,
 request.lastName);
        return new ResponseClass(greetingString);
}
```

To create a Lambda function with this handler, you must provide implementation of the input and output types as shown in the following Java example. The `HelloPojo` class defines the `RequestClass` and `ResponseClass` and the `handler` method.

```
package example;

import com.amazonaws.services.lambda.runtime.Context;

public class HelloPojo {
```

```
    // Define two classes/POJOs for use with Lambda function.
    public static class RequestClass {
        String firstName;
        String lastName;

        public String getFirstName() {
            return firstName;
        }

        public void setFirstName(String firstName) {
            this.firstName = firstName;
        }

        public String getLastName() {
            return lastName;
        }

        public void setLastName(String lastName) {
            this.lastName = lastName;
        }

        public RequestClass(String firstName, String lastName) {
            this.firstName = firstName;
            this.lastName = lastName;
        }

        public RequestClass() {
        }
    }

    public static class ResponseClass {
        String greetings;

        public String getGreetings() {
            return greetings;
        }

        public void setGreetings(String greetings) {
            this.greetings = greetings;
        }

        public ResponseClass(String greetings) {
            this.greetings = greetings;
        }

        public ResponseClass() {
        }

    }

   public static ResponseClass myHandler(RequestClass request, Context context){

       String greetingString = String.format("Hello %s, %s.", request.firstName,
 request.lastName);
        return new ResponseClass(greetingString);
    }
}
```

Note
The `get` and `set` methods are required in order for the POJOs to work with AWS Lambda's built in JSON serializer. The constructors that take no arguments are usually not required, however in this example we provided other constructors and therefore we need to explicitly provide the zero argument constructors.

You can upload this code as your Lambda function and test as follows:

- Using the preceding code, create a deployment package.
- Upload the deployment package to AWS Lambda and create your Lambda function. You can do this using the console or AWS CLI.
- Invoke the Lambda function manually using the console or the CLI. You can use provide sample JSON event data when you manually invoke your Lambda function. For example,

```
{ "firstName":"John", "lastName":"Doe" }
```

Follow instructions provided in the Getting Started exercise for Java. For more information, see Getting Started: Authoring AWS Lambda Code in Java (p. 88). Note the following differences:

- When you create a deployment package, don't forget the `aws-lambda-java-core` library dependency.
- When you create the Lambda function specify `example.HelloPojo::myHandler` (*package*.*class*::*method*) as the handler value.

Example: Using Stream for Handler Input/Output (Java)

If you do not want to use POJOs or if Lambda's serialization approach does not meet your needs, you can use the byte stream implementation. In this case, you can use the `InputStream` and `OutputStream` as the input and output types for the handler. An example hander function is shown:

```
public void handler(InputStream inputStream, OutputStream outputStream, Context
 context) {
   ...
}
```

Note that in this case the handler function uses parameters for both the request and response streams.

The following is a Lambda function example that implements the handler that uses `InputStream` and `OutputStream` types for the `input` and `output` parameters.

```
package example;

import java.io.InputStream;
import java.io.OutputStream;

import com.amazonaws.services.lambda.runtime.Context;

public class Hello {
    public static void handler(InputStream inputStream, OutputStream output
Stream, Context context) throws IOException {
        int letter;
        while((letter = inputStream.read()) != -1)
        {
            outputStream.write(Character.toUpperCase(letter));
```

```
        }
    }
}
```

You can do the following to test the code:

- Using the preceding code, create a deployment package.
- Upload the deployment package to AWS Lambda and create your Lambda function. You can do this using the console or AWS CLI.
- You can manually invoke the code by providing sample input. For example:

```
test
```

Follow instructions provided in the Getting Started exercise for Java. For more information, see Getting Started: Authoring AWS Lambda Code in Java (p. 88). Note the following differences:

- When you create a deployment package, don't forget the `aws-lambda-java-core` library dependency.
- When you create the Lambda function specify "example.Hello::handler" (*`package`*.*`class`*::*`method`*) as handler value.

Leveraging Predefined Interfaces for Creating Handler (Java)

Instead of loading handler method directly without implementing an interface (see Lambda Function Handler (Java) (p. 102)) for your Lambda function, you can implement one of the predefined interfaces, `RequestStreamHandler` or `RequestHandler` and provide implementation for the `handleRequest` method that the interfaces provide. You implement one of these interfaces depending on whether you want to use standard Java types or custom POJO types for your handler input/output (where AWS Lambda automatically serializes and deserializes the input and output to Match your data type), or customize the serialization using the `Stream` type.

> **Note**
> These interfaces are available in the `aws-lambda-java-core` library.

When you implement standard interfaces, they help you validate your method signature at compile time.

If you implement one of the interfaces, you specify *`package`*.*`class`* in your Java code as the handler when you create the Lambda function. For example, the following is the modified `create-function` CLI command from the getting started. Note that the `--handler` parameter specifies "example.Hello" value:

```
aws lambda create-function \
--region us-west-2 \
--function-name getting-started-lambda-function-in-java \
--zip-file fileb://deployment-package (zip or jar)
        path \
--role arn:aws:iam::account-id:role/lambda_basic_execution  \
--handler example.Hello \
--runtime java8 \
--timeout 15 \
--memory-size 512
```

The following sections provide examples of implementing these interfaces.

Example 1: Creating Handler with Custom POJO Input/Output (Leverage the RequestHandler Interface)

The example `Hello` class in this section implements the `RequestStreamHandler` interface. The interface defines `handleRequest()` method that takes in event data as input parameter of the `Request` type and returns an POJO object of the `Response` type:

```
public Response handleRequest(Request request, Context context) {
   ...
}
```

The `Hello` class with sample implementation of the `handleRequest()` method is shown. For this example, we assume event data consists of first name and last name.

```
package example;

import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.Context;

public class Hello implements RequestHandler<Request, Response> {

    public Response handleRequest(Request request, Context context) {
       String greetingString = String.format("Hello %s %s.", request.firstName,
 request.lastName);
        return new Response(greetingString);
    }
}
```

For example, if the event data in the `Request` object is:

```
{
  "firstName":"value1",
  "lastName" : "value2"
}
```

The method returns a `Response` object as follows:

```
{
  "greetings": "Hello value1 value2."
}
```

Next, you need to implement the `Request` and `Response` classes. You can use the following implementation for testing:

The Request class:

```
package example;

public class Request {
    String firstName;
    String lastName;

    public String getFirstName() {
```

```
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public Request(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public Request() {
    }
}
```

The Response class:

```
package example;

public class Response {
    String greetings;

    public String getGreetings() {
        return greetings;
    }

    public void setGreetings(String greetings) {
        this.greetings = greetings;
    }

    public Response(String greetings) {
        this.greetings = greetings;
    }

    public Response() {
    }
}
```

You can create a Lambda function from this code and test the end-to-end experience as follows:

- Using the preceding code, create a deployment package.
- Upload the deployment package to AWS Lambda and create your Lambda function.
- Test the Lambda function using either the console or CLI. You can specify any sample JSON data that conform to the getter and setter in your `Request` class, for example:

```
{
  "firstName":"John",
  "lastName" : "Doe"
}
```

The Lambda function will return the following JSON in response.

```
{
  "greetings": "Hello John, Doe."
}
```

Follow instructions provided in the getting started (see Getting Started: Authoring AWS Lambda Code in Java (p. 88)). Note the following differences:

- When you create a deployment package, don't forget the `aws-lambda-java-core` library dependency.
- When you create the Lambda function specify "example.Hello" (*`package`*.*`class`*) as handler value.

Example 2: Creating Handler with Stream Input/Output (Leverage the `RequestStreamHandler` Interface)

The `Hello` class in this example implements the `RequestStreamHandler` interface. The interface defines `handleRequest` method as follows:

```
public void handleRequest(InputStream inputStream, OutputStream outputStream,
Context context)
        throws IOException {
      ...
}
```

The `Hello` class with sample implementation of the `handleRequest()` handler is shown. The handler processes incoming event data (for example, a string "hello") by simply converting it to uppercase and return it.

```
package example;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

import com.amazonaws.services.lambda.runtime.RequestStreamHandler;
import com.amazonaws.services.lambda.runtime.Context;

public class Hello implements RequestStreamHandler {
   public void handleRequest(InputStream inputStream, OutputStream outputStream,
 Context context)
            throws IOException {
        int letter;
        while((letter = inputStream.read()) != -1)
        {
            outputStream.write(Character.toUpperCase(letter));
        }
```

```
    }
}
```

You can create a Lambda function from this code and test the end-to-end experience as follows:

- Use the preceding code to create deployment package.
- Upload the deployment package to AWS Lambda and create your Lambda function.
- Test the Lambda function using either the console or CLI. You can specify any sample string data, for example:

```
"test"
```

  The Lambda function will return "TEST" in response.

Follow instructions provided in the getting started (see Getting Started: Authoring AWS Lambda Code in Java (p. 88)). Note the following differences:

- When you create a deployment package, don't forget the `aws-lambda-java-core` library dependency.
- When you create the Lambda function specify "example.Hello" (*`package.class`*) as handler value.

The Context Object (Java)

You interact with AWS Lambda execution environment via the context parameter. The context object allows you to access useful information available within the Lambda execution environment. For example, you can use the context parameter to determine the CloudWatch log stream associated with the function, or use the clientContext property of the context object to learn more about the application calling the Lambda function (when invoked through the AWS Mobile SDK).

The context object properties are:

- `getMemoryLimitInMB()`: Memory limit, in MB, you configured for the Lambda function.
- `getFunctionName()`: Name of the Lambda function that is running.
- `getAwsRequestId()`: AWS request ID associated with the request. This is the ID returned to the client called the invoke(). You can use the request ID for any follow up enquiry with AWS support. Note that if AWS Lambda retries the function (for example, in a situation where the Lambda function processing Amazon Kinesis records throw an exception), the request ID remains the same.
- `getLogStreamName()`: The CloudWatch log stream name for the particular Lambda function execution. It can be null if the IAM user provided does not have permission for CloudWatch actions.
- `getLogGroupName()`: The CloudWatch log group name associated with the Lambda function invoked. It can be null if the IAM user provided does not have permission for CloudWatch actions.
- `getClientContext()`: Information about the client application and device when invoked through the AWS Mobile SDK. It can be null. Client context provides client information such as client ID, application title, version name, version code, and the application package name).
- `getIdentity()`: Information about the Amazon Cognito identity provider when invoked through the AWS Mobile SDK. It can be null.
- `getRemainingTimeInMillis()`: Remaining execution time till the function will be terminated, in milliseconds. At the time you create the Lambda function you set maximum time limit, at which time AWS Lambda will terminate the function execution. Information about the remaining time of function execution can be used to specify function behavior when nearing the timeout.
- `getLogger()`: Returns the Lambda logger associated with the Context object. For more information, see Logging (Java) (p. 114).

The following Java code snippet shows a handler function that prints some of the context information.

```
public static void handler(InputStream inputStream, OutputStream outputStream,
 Context context) {

  ...
        System.out.println("Function name: " + context.getFunctionName());
       System.out.println("Max mem allocated: " + context.getMemoryLimitInMB());

        System.out.println("Time remaining in milliseconds: " + context.getRe
mainingTimeInMillis());
       System.out.println("CloudWatch log stream name: " + context.getLogStream
Name());
        System.out.println("CloudWatch log group name: " + context.getLogGroup
Name());
}
```

Example: Using Context Object (Java)

The following Java code example shows how to use the `Context` object to retrieve runtime information of your Lambda function, while it is running.

```
package example;
import java.io.InputStream;
import java.io.OutputStream;
import com.amazonaws.services.lambda.runtime.Context;

public class Hello {
    public static void myHandler(InputStream inputStream, OutputStream output
Stream, Context context) {

        int letter;
        try {
            while((letter = inputStream.read()) != -1)
            {
                outputStream.write(Character.toUpperCase(letter));
            }
           Thread.sleep(3000); // Intentional delay for testing the getRemain
ingTimeInMillis() result.
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }

        // For fun, let us get function info using the context object.
        System.out.println("Function name: " + context.getFunctionName());
       System.out.println("Max mem allocated: " + context.getMemoryLimitInMB());

        System.out.println("Time remaining in milliseconds: " + context.getRe
mainingTimeInMillis());
       System.out.println("CloudWatch log stream name: " + context.getLogStream
Name());
        System.out.println("CloudWatch log group name: " + context.getLogGroup
Name());
```

```
    }
}
```

You can do the following to test the code:

- Using the preceding code, create a deployment package.
- Upload the deployment package to AWS Lambda to create your Lambda function. You can do this using the console or AWS CLI.
- To test your Lambda function use the "Hello World" **Sample event** that the Lambda console provides.

  You can type any string and the function will return the same string in uppercase. In addition, you will also get the useful function information provided by the `context` object.

Follow the instructions provided in the Getting Started exercise for Java. For more information, see Getting Started: Authoring AWS Lambda Code in Java (p. 88). Note the following differences:

- When you create a deployment package, don't forget the `aws-lambda-java-core` library dependency.
- When you create the Lambda function, specify `example.Hello::myHandler` (*`package`*`.`*`class`*`::`*`method`*) as the handler value.

Logging (Java)

You Lambda function can log information to CloudWatch. The following statements in your Lambda function code generate log entries:

- System.out()
- System.err()
- LambdaLogger.log()

We recommend using the `LambdaLogger` object to write logs to CloudWatch Logs. AWS Lambda treats each line returned by `System.out` and `System.err` as a separate event. This works well when each output line corresponds to a single log entry. When a conceptual log entry has multiple lines of output, AWS Lambda will attempt to parse them using line breaks to identify separate events. For example, `System.out.println("Hello \n world")` will log the two words as two separate event (`LambdaLogger.log()` will this as single event).

Each call to `LambdaLogger.log()` creates an event, provided within the event size, in CloudWatch logs. For information about CloudWatch limits, go to CloudWatch Limits in the Amazon CloudWatch Developer Guide.

You can find the logs that your Lambda function writes, as follows:

- Find logs in CloudWatch Logs.

  The `context` object (in the `aws-lambda-java-core` library) provides the `getLogStreamName()` and the `getLogGroupName()` methods. Using these methods, you can find the specific log stream where logs are written.
- If you invoke a Lambda function via the console, the invocation type is always `RequestResonse` (that is, synchronous execution), and the console displays the logs that the Lambda function writes using the `LambdaLogger` object. AWS Lambda also returns logs from `System.out` and `System.err` methods.
- If you invoke a Lambda function programmatically, you can add the `LogType` parameter to retrieve the last 4 KB of log data that is written to CloudWatch Logs. For more information, see Invoke (p. 178). AWS

Lambda returns this log information in the `x-amz-log-results` header in the response. If you use the AWS Command Line Interface to invoke the function, you can specify the `--log-type` parameter with value `Tail`.

Example: Writing Logs (Java)

The following Java code example shows how Lambda function can log information. The handler method (`myHandler`) uses `System.out`, `System.err`, and `LambdaLogger` object to write logs. For information about logging, see Logging (Java) (p. 114).

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;

public class Hello {
    public String myHandler(String name, Context context) {
        LambdaLogger logger = context.getLogger();
        // Write log to CloudWatch using LambdaLogger.
        logger.log("log data from Lambda logger");

        // System.out also generates log in CloudWatch but
        System.out.println("log data from stdout");

        System.err.println("log data from stderr.");

        // Return will include the log stream name so you can look
        // up the log later.
       return String.format("Hello %s. log stream = %s", name, context.getLog
StreamName());
    }
}
```

You can do the following to test the code:

- Using the preceding code, create a deployment package.
- Upload the deployment package to AWS Lambda to create your Lambda function. You can do this using the console or AWS CLI.
- To test your Lambda function use the "Hello World" **Sample event** that Lambda console provides.

  The handler code receives the sample event but does nothing with it. It only shows how to write logs.

Follow instructions provided in the Getting Started exercise for Java. For more information, see Getting Started: Authoring AWS Lambda Code in Java (p. 88). Note the following differences:

- When you create a deployment package, don't forget the `aws-lambda-java-core` library dependency.
- When you create the Lambda function, specify `example.Hello::myHandler` (*`package`*`.`*`class`*`::`*`method`*) as the handler value.

Exceptions (Java)

If your Lambda function throws exception, AWS Lambda recognizes the failure and serializes the exception information into JSON and returns it. An example error message is shown:

```
{
  "errorMessage": "Name John Doe is invalid. Exception occurred...",
  "errorType": "java.lang.Exception",
  "stackTrace": [
    "example.Hello.handler(Hello.java:9)",
    "sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)",
    "sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorIm
pl.java:62)",
    "sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessor
Impl.java:43)",
    "java.lang.reflect.Method.invoke(Method.java:497)"
  ]
}
```

Note that the stack trace is returned as the `stackTrace` JSON array of stack trace elements.

The method in which you get the error information back depends on the invocation type that you specified at the time you invoked the function:

- `RequestResponse` invocation type (that is, synchronous execution): In this case, you get the error message back.

  For example, if you invoke a Lambda function using the Lambda console, the `RequestResponse` is always the invocation type and the console displays the error information returned by AWS Lambda in the **Execution result** section as shown in the following image.

- `Event` invocation type (that is, asynchronous execution): In this case AWS Lambda does not return anything. Instead, it logs the error information in CloudWatch Logs and CloudWatch metrics.

Depending on the event source, AWS Lambda may retry the failed Lambda function. For example, if Amazon Kinesis is the event source for the Lambda function, AWS Lambda will retry the failed function until the Lambda function succeeds or the records in the stream expire.

AWS Lambda Example Walkthroughs (Java)

This section provides additional examples that demonstrate how to use Java.

Topics

- AWS Lambda Walkthrough 4: Handling Mobile User Application Events for Android (Java) (p. 124)

Important
We recommend you first review the information in Getting Started: Authoring AWS Lambda Code in Java (p. 88) and read Programming Model for Authoring Lambda Functions in Java (p. 101).

AWS Lambda Walkthrough 1: Process S3 Events (Java)

Suppose you have two buckets in Amazon S3. You store images (.jpg and .png objects) in one bucket ("*source*"). For each object created in the bucket, you want AWS Lambda to execute a Lambda function to create a thumbnail in the "*source*resized" bucket. You will use Amazon S3's bucket notification configuration feature to request Amazon S3 to publish object-created events to AWS Lambda. In the notification configuration, you will identify your Lambda function (called `CreateThumbnail`) that you want Amazon S3 to invoke. You will create the Lambda function in this exercise.

Important
We recommend you first review Getting Started: Authoring AWS Lambda Code in Java (p. 88)) and also see Programming Model for Authoring Lambda Functions in Java (p. 101).

This is an example of "push" model, where Amazon S3 detects an object created event and invokes Lambda function by passing in the event information.

The following is example Java code that reads incoming Amazon S3 event and creates a thumbnail. Note the implements the `RequestHandler` interface provided in the `aws-lambda-java-core` library. Therefore, at the time you create a Lambda function you specify the class as the handler (that is, `example.S3EventProcessorCreateThumbnail`). For more information about using interfaces to provide handler, see Leveraging Predefined Interfaces for Creating Handler (Java) (p. 108).

The `S3Event` type, the handler uses as the input type, is one of the predefined classes in the `aws-lambda-java-events` library that provides methods for you to easily read information from the incoming Amazon S3 event. The handler returns a string as output.

```
package example;

import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.RenderingHints;
import java.awt.image.BufferedImage;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.URLDecoder;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import javax.imageio.ImageIO;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.S3Event;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3Client;
import com.amazonaws.services.s3.event.S3EventNotification.S3EventNotification
```

```
Record;
import com.amazonaws.services.s3.model.GetObjectRequest;
import com.amazonaws.services.s3.model.ObjectMetadata;
import com.amazonaws.services.s3.model.S3Object;

public class S3EventProcessorCreateThumbnail implements
        RequestHandler<S3Event, String> {
    private static final float MAX_WIDTH = 100;
    private static final float MAX_HEIGHT = 100;
    private final String JPG_TYPE = (String) "jpg";
    private final String JPG_MIME = (String) "image/jpeg";
    private final String PNG_TYPE = (String) "png";
    private final String PNG_MIME = (String) "image/png";

    public String handleRequest(S3Event s3event, Context context) {
        try {
            S3EventNotificationRecord record = s3event.getRecords().get(0);

            String srcBucket = record.getS3().getBucket().getName();
            // Object key may have spaces or unicode non-ASCII characters.
            String srcKey = record.getS3().getObject().getKey()
                    .replace('+', ' ');
            srcKey = URLDecoder.decode(srcKey, "UTF-8");

            String dstBucket = srcBucket + "resized";
            String dstKey = "resized-" + srcKey;

           // Sanity check: validate that source and destination are different

            // buckets.
            if (srcBucket.equals(dstBucket)) {
                System.out
                        .println("Destination bucket must not match source
bucket.");
                return "";
            }

            // Infer the image type.
          Matcher matcher = Pattern.compile(".*\\.([^\\.]*)").matcher(srcKey);

            if (!matcher.matches()) {
                System.out.println("Unable to infer image type for key "
                        + srcKey);
                return "";
            }
            String imageType = matcher.group(1);
          if (!(JPG_TYPE.equals(imageType)) && !(PNG_TYPE.equals(imageType)))
 {
                System.out.println("Skipping non-image " + srcKey);
                return "";
            }

            // Download the image from S3 into a stream
            AmazonS3 s3Client = new AmazonS3Client();
            S3Object s3Object = s3Client.getObject(new GetObjectRequest(
                    srcBucket, srcKey));
            InputStream objectData = s3Object.getObjectContent();
```

```
            // Read the source image
            BufferedImage srcImage = ImageIO.read(objectData);
            int srcHeight = srcImage.getHeight();
            int srcWidth = srcImage.getWidth();
            // Infer the scaling factor to avoid stretching the image
            // unnaturally
            float scalingFactor = Math.min(MAX_WIDTH / srcWidth, MAX_HEIGHT
                    / srcHeight);
            int width = (int) (scalingFactor * srcWidth);
            int height = (int) (scalingFactor * srcHeight);

            BufferedImage resizedImage = new BufferedImage(width, height,
                    BufferedImage.TYPE_INT_RGB);
            Graphics2D g = resizedImage.createGraphics();
            // Fill with white before applying semi-transparent (alpha) images

            g.setPaint(Color.white);
            g.fillRect(0, 0, width, height);
            // Simple bilinear resize
            // If you want higher quality algorithms, check this link:
            // https://today.java.net/pub/a/today/2007/04/03/perils-of-image-
getscaledinstance.html
            g.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
                    RenderingHints.VALUE_INTERPOLATION_BILINEAR);
            g.drawImage(srcImage, 0, 0, width, height, null);
            g.dispose();

            // Re-encode image to target format
            ByteArrayOutputStream os = new ByteArrayOutputStream();
            ImageIO.write(resizedImage, imageType, os);
            InputStream is = new ByteArrayInputStream(os.toByteArray());
            // Set Content-Length and Content-Type
            ObjectMetadata meta = new ObjectMetadata();
            meta.setContentLength(os.size());
            if (JPG_TYPE.equals(imageType)) {
                meta.setContentType(JPG_MIME);
            }
            if (PNG_TYPE.equals(imageType)) {
                meta.setContentType(PNG_MIME);
            }

            // Uploading to S3 destination bucket
            System.out.println("Writing to: " + dstBucket + "/" + dstKey);
            s3Client.putObject(dstBucket, dstKey, is, meta);
            System.out.println("Successfully resized " + srcBucket + "/"
                   + srcKey + " and uploaded to " + dstBucket + "/" + dstKey);

            return "Ok";
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Amazon S3 invokes your Lambda function using the "Event" invocation type, where AWS Lambda executes the code asynchronously. What you return does not matter. However, in this case we are implementing the interface that requires that we specify a return type, so in this example the handler uses String as the return type.

You can do the following to test the code:

- Using the preceding code, create a deployment package. Make sure you add the following dependencies:

  `aws-lambda-java-core`

  `aws-lambda-java-events`
- Upload the deployment package to AWS Lambda to create your Lambda function. You can do this using the console or AWS CLI.
- On the Amazon S3 side, do the following:
  - Create two buckets (*`source`*, and "*`source`*resized")
  - Add notification configuration to *`source`* bucket request Amazon S3 to publish any object created event to the Lambda function you created.

  After the Lambda function executes, it writes logs to Amazon CloudWatch Logs. Because you are using the LambdaLogger to write logs, the console will display the log information.
- To test it, upload a .jpg or .png object to the *`source`* bucket and verify Lambda function created a thumbnail in the *`source`*`resized` bucket.

Follow instructions provided in the Getting Started exercise for Java to create your Lambda function. For more information, see Getting Started: Authoring AWS Lambda Code in Java (p. 88). Note the following differences:

- When you create a deployment package, don't forget the two libraries (`aws-lambda-java-core` and `aws-lambda-java-events`) you add as dependencies.
- When you create the Lambda function (called `CreateThumbnail`) specify the class `example.S3EventProcessorCreateThumbnail (`*`package`*`.`*`class`*`)` as the handler.
- Use the **S3 execution role** when creating the function in the console. This role has the necessary permissions to access Amazon S3 resources.

For instructions on adding notification configuration to a bucket, go to Enabling Event Notifications in the *Amazon Simple Storage Service Console User Guide*.

AWS Lambda Walkthrough 2: Process Kinesis Events (Java)

The Java example provided in this walkthrough processes events published by Amazon Kinesis.

> **Important**
> We recommend you first try the Getting Started exercises for Java in Getting Started: Authoring AWS Lambda Code in Java (p. 88) and also read the programming model section in Programming Model for Authoring Lambda Functions in Java (p. 101).

AWS Lambda polls the Amazon Kinesis stream and, when new records are found, it invokes your Lambda function. The Lambda function receives the records from the Kinesis stream as input, and in this example it writes to CloudWatch logs.

> **Note**
> This is an example of "pull" model, where AWS Lambda polls an Amazon Kinesis stream and invokes Lambda function when it finds a new record in the stream.

In the code, `recordHandler` is the handler. The handler uses predefined `KinesisEvent` class defined in the `aws-lambda-java-events` library.

```
package example;

import java.io.IOException;

import com.amazonaws.services.lambda.runtime.events.KinesisEvent;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent.KinesisEventRe
cord;

public class ProcessKinesisEvents {
    public void recordHandler(KinesisEvent event) throws IOException {
        for(KinesisEventRecord rec : event.getRecords()) {
           System.out.println(new String(rec.getKinesis().getData().array()));

        }
    }
}
```

If the handler returns normally, Lambda considers the input batch of records as processed successfully and begins reading new records in the stream. If the handler throws an exception, Lambda considers the input batch of records as not processed and invokes the function with the same batch of records again.

You can do the following to test the code:

- Using the preceding code (in a file named ProcessKinesisEvents.java), create a deployment package. Make sure you add the following dependencies:

  `aws-lambda-java-core`

  `aws-lambda-java-events`
- Upload the deployment package to AWS Lambda to create your Lambda function. You can do this using the console or AWS CLI.
- You can test this function without having to create an Amazon Kinesis stream. The AWS Lambda console provides sample Amazon Kinesis event data that you can use to invoke the function.

  After the Lambda function executes, it reads the incoming event, reads data from the Amazon Kinesis records, and writes logs to Amazon CloudWatch Logs. The same log also appears in the console **Execution logs** section.

Follow the instructions provided in the Getting Started exercise for Java to create your Lambda function. For more information, see Getting Started: Authoring AWS Lambda Code in Java (p. 88). Note the following differences:

- When you create a deployment package, don't forget the two libraries (`aws-lambda-java-core` and `aws-lambda-java-events`) you add as dependencies.
- When you create the Lambda function specify `example.ProcessKinesisEvents::recordHandler` (*`package`*`.`*`class`*`::`*`handler`*) as the handler value.
- Use the **Kinesis execution role** when you create the function in the console. The console associates the following access policy with this role grants the necessary permissions to access the Amazon Kinesis stream.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
```

```
      "Action": [
        "lambda:InvokeFunction"
      ],
      "Resource": [
        "*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "kinesis:GetRecords",
        "kinesis:GetShardIterator",
        "kinesis:DescribeStream",
        "kinesis:ListStreams",
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents"
      ],
      "Resource": "*"
    }
  ]
}
```

After testing the function in the console you can create an Amazon Kinesis stream and test the end-to-end experience. For instructions about how to create an Amazon Kinesis stream and add records to it, see Step 2: Configure an Amazon Kinesis Stream as the Event Source Using the Console (Node.js) (p. 26).

AWS Lambda Walkthrough 3: Process Amazon DynamoDB Events (Java)

Topics

Overview

The Java example provided in this walkthrough processes events published by Amazon DynamoDB.

> **Important**
> We recommend you first try the Getting Started exercises for Java in Getting Started: Authoring AWS Lambda Code in Java (p. 88) and also read the programming model section in Programming Model for Authoring Lambda Functions in Java (p. 101).

AWS Lambda polls the DynamoDB stream and, when new records are found, it invokes your Lambda function. The Lambda function receives the records from the DynamoDB stream as input. In this example, the Lambda function processes the event by writing some of the event data to the CloudWatch logs.

> **Note**
> This is an example of *pull* model, where AWS Lambda polls an Amazon DynamoDB stream and invokes Lambda function when it finds a new record in the stream. For more information, see The Pull/Push Event Models (p. 5).

In the code, `handleRequest` is the handler that AWS Lambda invokes and provides event data. The handler uses the predefined `DynamodbEvent` class, which is defined in the `aws-lambda-java-events` library.

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent.DynamodbStream
Record;

public class DDBEventProcessor implements
        RequestHandler<DynamodbEvent, String> {

    public String handleRequest(DynamodbEvent ddbEvent, Context context) {
        LambdaLogger logger = context.getLogger();
        for (DynamodbStreamRecord record : ddbEvent.getRecords()){
           logger.log(record.getEventID() + "\n");
           logger.log(record.getEventName() + "\n");
           logger.log(record.getDynamodb().toString() + "\n");

        }
        return "Successfully processed " + ddbEvent.getRecords().size() + "
records.";
    }
}
```

If the handler returns normally, Lambda considers the input batch of records as processed successfully and begins reading new records in the stream. If the handler throws an exception, Lambda considers the input batch of records as not processed and invokes the function with the same batch of records again.

After the Lambda function executes, it reads the incoming event, reads data from the Amazon DynamoDB records, and writes logs to Amazon CloudWatch Logs. The same log also appears in the console **Execution logs** section.

Setting Up the Walkthrough

Use the following steps to create a Lambda function, create a DynamoDB table with streams enabled, and create event source mapping in AWS Lambda to associate the stream with the Lambda function.

1. Create a Lambda function and specify sample DynamoDB events provided by the AWS Lambda console.

   Follow the instructions provided in the Getting Started: Authoring AWS Lambda Code in Java (p. 88). As you follow the steps, use the following information:

   - When you create a deployment package, don't forget to add the two libraries (`aws-lambda-java-core` and `aws-lambda-java-events`) as dependencies.
   - When you create the Lambda function:
     - Specify `example.DDBEventProcessor` (*`package.class`*) as the handler value. Note that this handler string did not include the name of the handler method because your class implements the `RequestHandler` interface.
     - Specify the **DynamoDB event stream role** when you create the function in the console. The console associates the following access policy with this role that grants AWS Lambda necessary permissions to access the Amazon DynamoDB stream.

2. Test the end-to-end experience.

   Follow the instructions provided in the Step 3: Add an Event Source (DynamoDB Streams) and Test (p. 59) to do the following:

   - Create a DynamoDB table with streams enabled.
   - Associate the DynamoDB stream with the Lambda function by adding an event source mapping in AWS Lambda.

     Note
     You can add the mapping using the Lambda console, DynamoDB console, or using the AWS CLI. Instructions in Step 3: Add an Event Source (DynamoDB Streams) and Test (p. 59) use the AWS CLI to create event source mapping, but you can do the same using the Lambda console or the DynamoDB console.

AWS Lambda begins polling the stream and when AWS Lambda detects new records on the stream, it invokes your Lambda function by passing the event data to your Lambda function handler.

AWS Lambda Walkthrough 4: Handling Mobile User Application Events for Android (Java)

Scenario

In this walkthrough, you will create a simple Android mobile user application. The primary purpose of this walkthrough is to show you how to hook up various components to enable an Android mobile application to invoke a Lambda function and process response.

The mobile application processes events by invoking a Lambda function called `LambdaJavaExampleAndroidBackend` using the AWS Mobile SDK for Android, and passing the event data to the function. The application events are not complex and the event data consists of a name (first name and last name) as shown:

```
{ firstName: "value1", lastName: "value2" }
```

The Lambda function then processes the event and sends back a response.

Use the following Java code to create the `LambdaJavaExampleAndroidBackend` function in AWS Lambda. In the code, the `handler` (`myHandler`) uses the `RequestClass` and `ResponseClass` types for the input and output and code provides implementation for these types.

Important
You will use the same classes (`POJOs`) to handle the input and output data when you create the sample mobile application in the next section.

```
package example;

import com.amazonaws.services.lambda.runtime.Context;

public class HelloPojo {

    // Define two classes/POJOs for use with Lambda function.
    public static class RequestClass {
```

```
        String firstName;
        String lastName;

        public String getFirstName() {
            return firstName;
        }

        public void setFirstName(String firstName) {
            this.firstName = firstName;
        }

        public String getLastName() {
            return lastName;
        }

        public void setLastName(String lastName) {
            this.lastName = lastName;
        }

        public RequestClass(String firstName, String lastName) {
            this.firstName = firstName;
            this.lastName = lastName;
        }

        public RequestClass() {
        }
    }

    public static class ResponseClass {
        String greetings;

        public String getGreetings() {
            return greetings;
        }

        public void setGreetings(String greetings) {
            this.greetings = greetings;
        }

        public ResponseClass(String greetings) {
            this.greetings = greetings;
        }

        public ResponseClass() {
        }

    }

   public static ResponseClass myHandler(RequestClass request, Context context){

       String greetingString = String.format("Hello %s, %s.", request.firstName,
 request.lastName);
        return new ResponseClass(greetingString);
    }
}
```

Note
The way that the mobile application invokes a Lambda function as shown in this walkthrough is an example of the AWS Lambda *request-response* model in which an application invokes a Lambda function and receives a response in real time. For more information, see The Pull/Push Event Models (p. 5).

Implementation Summary

Note the following about the mobile application:

- The mobile application must have valid security credentials and permissions to invoke a Lambda function.

  The mobile application in this walkthrough uses the Amazon Cognito service to manage user identities, authentication, and permissions. As part of the application setup, you create an Amazon Cognito identity pool to store user identities and define permissions. For more information, see Amazon Cognito.
- This example mobile application does not require a user to log in.

  A mobile application can require its users to log in using public identity providers such as Amazon and Facebook. The scope of this walkthrough is limited and assumes that the mobile application users are unauthenticated. It is essential that you understand this before you create an Amazon Cognito identity pool and configure user permissions.

For this walkthrough, configure the Amazon Cognito identity pool as follows:

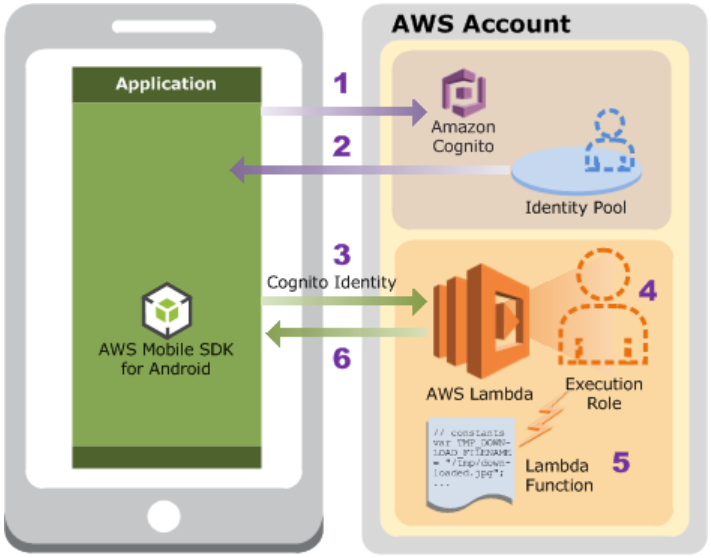
- Enable access for unauthenticated identities.

  In this walkthrough, the mobile application users are unauthenticated (within the mobile application), so Amazon Cognito provides a unique identifier and temporary AWS credentials for these users to invoke the Lambda function.
- Add permission to invoke the Lambda function in the access policy associated with the IAM role for unauthenticated users.

  An identity pool has two associated IAM roles, one for authenticated and one for unauthenticated application users. Depending on the application user, Amazon Cognito assumes one of the roles when it generates temporary credentials for the user. In this example, Amazon Cognito assumes the role for unauthenticated users to obtain temporary credentials. Using the temporary credentials, the application user can then invoke the Lambda function.

  The access policy associated with the IAM role determines what the mobile application user can do when using the temporary credentials. In this walkthrough, you update the policy to allow permission to invoke the `LambdaJavaExampleAndroidBackend` function.

The following diagram illustrates the application flow:



- Step 1: The mobile application sends the request to Amazon Cognito, and Amazon Cognito uses the identity pool ID in the requests.
- Step 2: The application receives temporary security credentials from Amazon Cognito.

  Amazon Cognito assumes the role associated with the identity pool and generates temporary credentials. What the application can do with the temporary credentials is limited by the access policy associated with the role. The AWS SDK can cache the temporary credentials so that the application does not send a request to Amazon Cognito each time it needs to invoke a Lambda function.
- Step 3: The mobile application invokes the Lambda function using temporary credentials (Cognito Identity).

AWS Lambda executes the function and responds immediately (in real time) with output as follows:

- Step 4: AWS Lambda assumes the execution role to execute your Lambda function on your behalf.
- Step 5: The Lambda function executes.
- Step 6: AWS Lambda returns results to the mobile application.

Now you are ready to try the steps.

Note that after the initial preparation, the walkthrough is divided into two main sections:

- First, you perform the necessary setup to create a Lambda function. Instead of creating an event source (the Android mobile application), you invoke the Lambda function manually using sample event data. This intermediate testing verifies that the function works.
- Second, you create an Amazon Cognito identity pool to manage authentication and permissions, and create the example Android application. When you run the Android application, it creates sample events and invokes your Lambda function.

Next Step

Step 1: Preparing for the Walkthrough (p. 128)

Step 1: Preparing for the Walkthrough

We recommend that you do not use the root credentials of your AWS account. Instead, create an administrator user in your account and use the administrator user credentials in setting up the walkthrough. If you already have an administrator user, you can skip this step.

For instructions to create an administrator user, see Set Up an AWS Account and Create an Administrator User (p. 12).

> **Note**
> The walkthrough assumes you are creating a Lambda function and an Amazon Cognito identity pool in the `us-west-2` region. If you want to use a different AWS region, make sure you create these resources in the same region. You also need to update the example mobile application code by providing the specific region that you want to use.

Next Step

Step 2: Create and Test the Lambda Function (p. 128)

Step 2: Create and Test the Lambda Function

Use the following steps to create a Lambda function and test it using sample event data. You will use the AWS Lambda console to test the Lambda function.

Topics

Step 2.1: Create the Lambda Function

Follow the instructions provided in the Getting Started: Authoring AWS Lambda Code in Java (p. 88).

Use the following Java code to create your Lambda function (`ExampleAndroidEventProcessor`).

```
package example;

import com.amazonaws.services.lambda.runtime.Context;

public class HelloPojo {

    // Define two classes/POJOs for use with Lambda function.
    public static class RequestClass {
        String firstName;
        String lastName;

        public String getFirstName() {
            return firstName;
        }

        public void setFirstName(String firstName) {
            this.firstName = firstName;
        }

        public String getLastName() {
            return lastName;
```

```
        }

        public void setLastName(String lastName) {
            this.lastName = lastName;
        }

        public RequestClass(String firstName, String lastName) {
            this.firstName = firstName;
            this.lastName = lastName;
        }

        public RequestClass() {
        }
    }

    public static class ResponseClass {
        String greetings;

        public String getGreetings() {
            return greetings;
        }

        public void setGreetings(String greetings) {
            this.greetings = greetings;
        }

        public ResponseClass(String greetings) {
            this.greetings = greetings;
        }

        public ResponseClass() {
        }

    }

   public static ResponseClass myHandler(RequestClass request, Context context){

       String greetingString = String.format("Hello %s, %s.", request.firstName,
 request.lastName);
        context.getLogger().log(greetingString);
        return new ResponseClass(greetingString);
    }
}
```

As you follow the steps, use the following information:

- When you create a deployment package, remember to add the `aws-lambda-java-core` library as the dependency.
- When you create the Lambda function:
  - Specify `ExampleAndroidEventProcessor` as the function name. The same name is also used in the Android application in the next section.

    Specify `example.HelloPojo::myHandler` (*`package`*.*`class`*::*`handler`*) as the `handler` value.
  - Specify the **Basic execution role** when you create the function in the console.

Step 2.2: Test the Lambda Function

Now you are ready to test the Lambda function by manually invoking it. For step-by-step instructions, see Step 3: Test the Lambda Function (p. 92). Use the following sample event data:

```
{
  "firstName": "first-name",
  "lastName": "last-name"
}
```

Next Step

Step 3: Create an Amazon Cognito Identity Pool (p. 130)

Step 3: Create an Amazon Cognito Identity Pool

In this section, you create an Amazon Cognito identity pool. The identity pool has two IAM roles. You update the IAM role for unauthenticated users and grant permission to execute the `LambdaJavaExampleAndroidBackend` Lambda function.

For more information about IAM roles, see IAM Roles (Delegation and Federation) in the *IAM User Guide*. For more information about Amazon Cognito services, go to the Amazon Cognito product detail page.

To create an identity pool

1. Using the IAM User Sign-In URL, sign in to the Amazon Cognito console as **`adminuser`**.
2. Create a new identity pool called `JavaFunctionAndroidEventHandlerPool`. Before you follow the procedure to create an identity pool, note the following:

   - The identity pool you are creating must allow access to unauthenticated identities because our example mobile application does not require a user log in (the application users are unauthenticated). In this case, select the **`Enable access to unauthenticated identities`** option.
   - The unauthenticated application users need permission to invoke the Lambda function. To enable this, add the following statement to allow the `lambda:InvokeFunction` action for the specific Lambda function.

     ```
     {
             "Effect": "Allow",
             "Action": [
                 "lambda:InvokeFunction"
             ],
             "Resource": [
                "arn:aws:lambda:us-west-2:account-id:function:Lambda
     JavaExampleAndroidBackend"
             ]
     }
     ```

     The `Resource` ARN identifies your Lambda function to which you are granting the `lambda:InvokeFunction` action and `LambdaJavaExampleAndroidBackend` is the Lambda function name that you created in the preceding step.

     Update the access policy when the identity pool is being created. The resulting policy will be as follows:

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Effect":"Allow",
         "Action":[
            "mobileanalytics:PutEvents",
            "cognito-sync:*"
         ],
         "Resource":[
            "*"
         ]
      },
      {
         "Effect":"Allow",
         "Action":[
            "lambda:invokefunction"
         ],
         "Resource":[
            "arn:aws:lambda:us-west-2:account-id:function:Lambda
JavaExampleAndroidBackend"
         ]
      }
   ]
}
```

Note
After creating the identity pool, go to the IAM console, find the role for unauthenticated users, and edit the access policy. Make sure you write down the IAM role name for the unauthenticated users so that you can search for it in the IAM console.

For instructions about how to create an identity pool, log in to the Amazon Cognito console and follow the **New Identity Pool** wizard.

Write down the identity pool ID. You specify this ID in your mobile application. The application uses this ID when it sends request to Amazon Cognito to request for temporary security credentials.

Next Step

Step 4: Create a Mobile User Application for Android (p. 131)

Step 4: Create a Mobile User Application for Android

Now you can create a simple Android mobile application that generates events and invokes Lambda functions by passing the event data as parameters.

The following instructions have been verified using Android studio.

1. Create a new Android project called `AndroidEventGenerator` using the following configuration:
   - Select the **Phone and Tablet** platform.
   - Choose **Blank Activity**.
2. In the build.gradle (`Module:app`) file, add the following in the `dependencies` section:

```
compile 'com.amazonaws:aws-android-sdk-core:2.2.+'
compile 'com.amazonaws:aws-android-sdk-lambda:2.2.+'
```

3. Build the project so that the required dependencies are downloaded, as needed.
4. In the Android application manifest (`AndroidManifest.xml`), add the following permissions so that your application can connect to the Internet. You can add them just before the `</manifest>` end tag.

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

5. In `MainActivity`, add the following imports:

```
import com.amazonaws.mobileconnectors.lambdainvoker.*;
import com.amazonaws.auth.CognitoCachingCredentialsProvider;
import com.amazonaws.regions.Regions;
```

6. In the `package` section, specify the appropriate package name (as shown below *`com.example.....lambdaeventgenerator`*). Add a new class called `RequestClass`, instances of which act as the POJO (Plain Old Java Object) for event data which consists of first and last name.

   Note that the POJO is same as the the POJO you created in your Lambda function in the preceding section.

```
package com.example....lambdaeventgenerator;
public class RequestClass {
    String firstName;
    String lastName;

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public RequestClass(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public RequestClass() {
    }
}
```

7. Create another class, `ResponseClass`.

```
package com.example....lambdaeventgenerator;
public class ResponseClass {
    String greetings;

    public String getGreetings() {
        return greetings;
    }

    public void setGreetings(String greetings) {
        this.greetings = greetings;
    }

    public ResponseClass(String greetings) {
        this.greetings = greetings;
    }

    public ResponseClass() {
    }
}
```

8. In the specified package (as shown below *com.example.....lambdaeventgenerator*), create an interface called `MyInterface` for invoking the `LambdaJavaExampleAndroidBackend` Lambda function.

   Note
   The `@LambdaFunction` annotation in the code maps the specific client method to the same-name Lambda function. For more information about this annotation, go to AWS Lambda in the *AWS SDK for Android Developer Guide*.

```
package com.example.....lambdaeventgenerator;
import com.amazonaws.mobileconnectors.lambdainvoker.LambdaFunction;
public interface MyInterface {

    /**
     * Invoke the Lambda function "LambdaJavaExampleAndroidBackend".
     * The function name is the method name.
     */
    @LambdaFunction
     ResponseClass LambdaJavaExampleAndroidBackend(RequestClass request);

}
```

9. To keep the application simple, we are going to add code to invoke the Lambda function in the `onCreate()` event handler. In `MainActivity`, add the following code toward the end of the `onCreate()` code.

```
// Create an instance of CognitoCachingCredentialsProvider
CognitoCachingCredentialsProvider cognitoProvider = new CognitoCachingCreden
tialsProvider(
       this.getApplicationContext(), "identity-pool-id", Regions.US_WEST_2);

// Create LambdaInvokerFactory, to be used to instantiate the Lambda proxy.
LambdaInvokerFactory factory = new LambdaInvokerFactory(this.getApplication
Context(),
        Regions.US_WEST_2, cognitoProvider);
```

```
// Create the Lambda proxy object with a default Json data binder.
// You can provide your own data binder by implementing
// LambdaDataBinder.
final MyInterface myInterface = factory.build(MyInterface.class);

RequestClass request = new RequestClass("John", "Doe");
// The Lambda function invocation results in a network call.
// Make sure it is not called from the main thread.
new AsyncTask<RequestClass, Void, ResponseClass>() {
    @Override
    protected ResponseClass doInBackground(RequestClass... params) {
        // invoke "echo" method. In case it fails, it will throw a
        // LambdaFunctionException.
        try {
            return myInterface.LambdaJavaExampleAndroidBackend(params[0]);
        } catch (LambdaFunctionException lfe) {
            Log.e("Tag", "Failed to invoke echo", lfe);
            return null;
        }
    }

    @Override
    protected void onPostExecute(ResponseClass result) {
        if (result == null) {
            return;
        }

        // Do a toast
        Toast.makeText(MainActivity.this, result.getGreetings(),
Toast.LENGTH_LONG).show();
    }
}.execute(request);
```

10. Run the code and verify as follows:
    - The `Toast.makeText()` displays the response returned.
    - Verify that CloudWatch Logs shows the log created by the Lambda function. It should show the event data (first name and last name). You can also verify this in the AWS Lambda console.

Troubleshooting and Monitoring AWS Lambda Functions with Amazon CloudWatch

AWS Lambda automatically monitors Lambda functions on your behalf, reporting metrics through Amazon CloudWatch. To help you monitor your code as it executes, Lambda automatically tracks the number of requests, the latency per request, and the number of requests resulting in an error and publishes the associated CloudWatch metrics. You can leverage these metrics to set CloudWatch custom alarms. For more information about CloudWatch, see the Amazon CloudWatch Developer Guide.

You can view request rates and error rates for each of your Lambda functions by using the AWS Lambda console, the CloudWatch console, and other Amazon Web Services (AWS) resources. The following topics describe Lambda CloudWatch metrics and how to access them.

- Accessing Amazon CloudWatch Metrics for AWS Lambda (p. 137)
- AWS Lambda Metrics (p. 139)

You can insert logging statements into your code to help you validate if your code is working as expected. Lambda automatically integrates with Amazon CloudWatch Logs and pushes all logs from your code to a CloudWatch Logs group associated with a Lambda function (Lambda/<function name>). To learn more about log groups and accessing them through the CloudWatch console, see the Monitoring System, Application, and Custom Log Files in the *Amazon CloudWatch Developer Guide*.

The following topic describes how to access CloudWatch log entries.

- Accessing Amazon CloudWatch Logs for AWS Lambda (p. 138)

> **Note**
> If your Lambda function code is executing but you don't see any log data being generated after several minutes, this could mean your *execution role* for the Lambda function did not grant permissions to write log data to CloudWatch Logs. For information about how to make sure that you have set up the *execution role* correctly to give these permissions, see Execution Permissions (p. 8).

AWS Lambda Troubleshooting Scenarios

This sections describes examples of how to monitor and troubleshoot your Lambda functions using the logging and monitoring capabilities of CloudWatch.

Troubleshooting Scenario 1: Lambda function not working as expected

In this scenario, you have just finished AWS Lambda Walkthrough 2: Handling Amazon S3 Events Using the AWS CLI (Node.js) (p. 43). However, the Lambda function you created to upload a thumbnail image to Amazon S3 when you create an S3 object is not working as expected. When you upload objects to Amazon S3, you see that the thumbnail images are not being uploaded. You can troubleshoot this issue in the following ways.

To determine why your Lambda function is not working as expected

1. Check if your code is working correctly. An increased error rate would indicate that it is not.

   You can test your code locally as you would any other Node.js function, or you can test it within Lambda by using the "test invoke" capability on the console, or you can use the AWS CLI `asyncInvoke` command. Each time the code is executed in response to an event, it writes a log entry into the log group associated with a Lambda function, which is Lambda/<function name>.

   The following are some errors that might show up in the logs:

   - If you see a stack trace in your log, there is probably an error in your code. Review your code and debug the error the stack trace refers to.
   - If you see a "permissions denied" error in the log, the IAM role you have provided as an *execution role* may not have the necessary permissions. Check if the IAM role has all the necessary permissions to access any AWS resources your code references. To ensure that you have correctly set up the *execution role*, see Execution Permissions (p. 8).
   - If you see a "timeout exceeded" error in the log, your timeout setting exceeds the run time of your function code. This may be because the timeout is too low, or the code is taking too long to execute.
   - If you see a "memory exceeded" error in the log, your memory setting is too low. Set it to a higher value, for memory size limits see CreateFunction (p. 160). Changing the memory setting can change how you are charged for duration. For information about pricing, see AWS Lambda.

2. Check if your Lambda function is receiving requests.

   Even if your function code is working as expected and responding correctly to test invokes, the function may not be receiving requests from Amazon S3. If Amazon S3 is able to invoke the function, you should see an increase in your CloudWatch requests metric. If you do not see an increase in your CloudWatch requests, check the access policy associated with the function.

Troubleshooting Scenario 2: Increased latency in Lambda function execution

In this scenario, you have just finished AWS Lambda Walkthrough 2: Handling Amazon S3 Events Using the AWS CLI (Node.js) (p. 43). However, the Lambda function you created to upload a thumbnail image to Amazon S3 when you create an S3 object is not working as expected. When you upload objects to Amazon S3, you can see that the thumbnail images are being uploaded, but your code is taking much longer to execute than expected. You can troubleshoot this issue in a couple of different ways. For

example, you could monitor the "latency" CloudWatch metric for the Lambda function to see if the latency is increasing. Or you could see an increase in the "errors" CloudWatch metric for the Lambda function, which might be due to timeout errors.

To determine why there is increased latency in the execution of a Lambda function

1. Test your code with different memory settings.

   If your code is taking too long to execute, it could be that it does not have enough compute resources to execute its logic. Try increasing the memory allocated to your function and testing the code again, using the Lambda console test invoke functionality. You can see the memory used, code execution time, and memory allocated in the function log entries. Changing the memory setting can change how you are charged for duration. For information about pricing, see AWS Lambda.
2. See where the execution bottleneck is using logs.

   You can test your code locally as you would any other Node.js function, or you can test it within Lambda using the "test invoke" capability on the Lambda console, or the `asyncInvoke` command by using AWS CLI. Each time the code is executed in response to an event, it writes a log entry into the log group associated with a Lambda function, which is named (Lambda/<function name>). Add logging statements around various parts of your code, such as callouts to other services, to see how much time is being spent in executing different parts of your code.

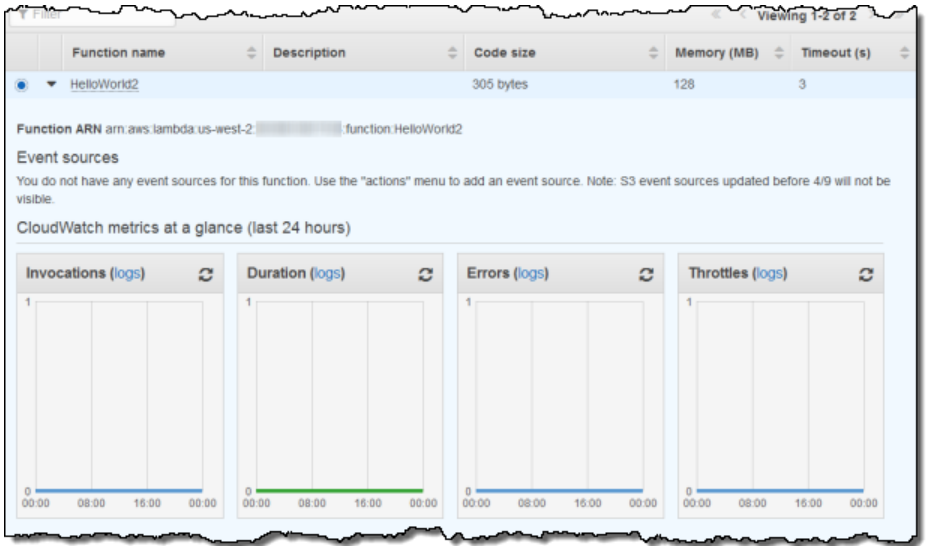
Accessing Amazon CloudWatch Metrics for AWS Lambda

AWS Lambda automatically monitors functions on your behalf, reporting metrics through Amazon CloudWatch. These metrics include total requests, latency, and error rates. For more information about Lambda metrics, see AWS Lambda Metrics (p. 139). For more information about CloudWatch, see the Amazon CloudWatch Developer Guide.

You can monitor metrics for Lambda and view logs by using the Lambda console, the CloudWatch console, the AWS CLI, or the CloudWatch API. The following procedures show you how to access metrics using these different methods.

To access metrics using the Lambda console

1. Sign in to the AWS Management Console and open the Lambda console.
2. If you have not created a Lambda function before, go to Getting Started: Authoring AWS Lambda Code in Node.js (p. 15).
3. On the **Lambda: Function List** page, click the radio button or the triangle to the left of a function name to expand the function details.



A graphical representation of the metrics for the Lambda function are shown.

4. Click **CloudWatch logs** next to a metric name to view the log for the metric.

To access metrics using the CloudWatch console

1. Open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. From the navigation bar, select a region.
3. In the navigation pane, click **Metrics**.
4. In the **CloudWatch Metrics by Category** pane, select **Lambda Metrics**.
5. (Optional) In the graph pane, select a statistic and a time period, and then create a CloudWatch alarm using these settings.

To access metrics using the AWS CLI

Use the list-metrics and get-metric-statistics commands.

To access metrics using the CloudWatch CLI

Use the mon-list-metrics and mon-get-stats commands.

To access metrics using the CloudWatch API

Use the `ListMetrics` and `GetMetricStatistics` operations.

Accessing Amazon CloudWatch Logs for AWS Lambda

AWS Lambda automatically monitors Lambda functions on your behalf, reporting metrics through Amazon CloudWatch. To help you troubleshoot failures in a function, Lambda logs all requests handled by your function and also automatically stores logs generated by your code through Amazon CloudWatch Logs.

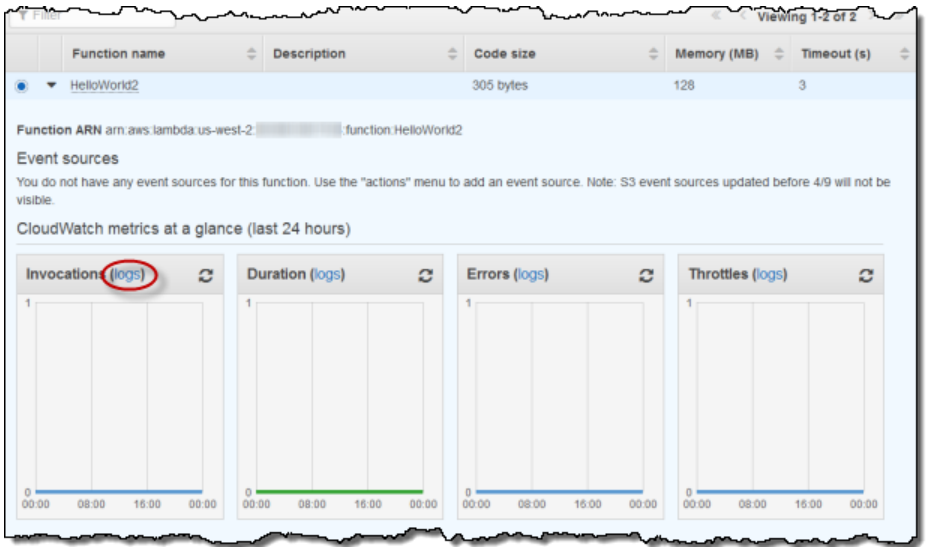
You can insert logging statements into your code to help you validate if your code is working as expected. Lambda automatically integrates with CloudWatch Logs and pushes all logs from your code to a

CloudWatch Logs group associated with a Lambda function, which is named Lambda/<function name>. To learn more about log groups and accessing them through the CloudWatch console, see the Monitoring System, Application, and Custom Log Files in the *Amazon CloudWatch Developer Guide*.

You can view logs for Lambda by using the Lambda console, the CloudWatch console, the AWS CLI, or the CloudWatch API. The following procedure show you how to view the logs by using the Lambda console.

To view logs using the Lambda console

1. Sign in to the AWS Management Console and open the Lambda console.
2. If you have not created a Lambda function before, go to Getting Started: Authoring AWS Lambda Code in Node.js (p. 15).
3. On the **Lambda: Function List** page, click the radio button or the triangle to the left of a function name to expand the function details.



   A graphical representation of the metrics for the Lambda function are shown.
4. Click **logs** next to a metric name to view the log for the metric.

For more information on accessing CloudWatch Logs, see the following guides:

- Amazon CloudWatch Developer Guide
- Amazon CloudWatch Logs API Reference
- Amazon CloudWatch Developer Guide Monitoring Log Files

AWS Lambda Metrics

This topic describes the AWS Lambda namespace, metrics, and dimensions. AWS Lambda automatically monitors functions on your behalf, reporting metrics through Amazon CloudWatch (CloudWatch). These metrics include total invocations, errors, duration, and throttles.

CloudWatch is basically a metrics repository. A metric is the fundamental concept in CloudWatch and represents a time-ordered set of data points. You or AWS products publish metric data points into CloudWatch and you retrieve statistics about those data points as an ordered set of time-series data.

Metrics are uniquely defined by a name, a namespace, and one or more dimensions. Each data point has a time stamp, and (optionally) a unit of measure. When you request statistics, the returned data stream is identified by namespace, metric name, and dimension. For more information about CloudWatch, see the *Amazon CloudWatch Developer Guide*.

AWS Lambda CloudWatch Metrics

The AWS Lambda namespace for CloudWatch is **AWS/Lambda**.

The following metrics are available from the AWS Lambda service.

Metric	Description
`Invocations`	Measures the number of times a function is invoked in response to an event or invocation API call. This replaces the deprecated RequestCount metric. This includes successful and failed invocations, but does not include throttled attempts. This equals the billed requests for the function. Note that AWS Lambda only sends these metrics to CloudWatch if they have a nonzero value. Units: Count
`Errors`	Measures the number of invocations that failed due to errors in the function (response code 4XX). This replaces the deprecated ErrorCount metric. Failed invocations may trigger a retry attempt that succeeds. This includes: • Handled exceptions (e.g., context.fail(error)) • Unhandled exceptions causing the code to exit • Out of memory exceptions • Timeouts • Permissions errors This does **not** include invocations that fail due to invocation rates exceeding default concurrent limits (error code 429) or failures due to internal service errors (error code 500). Units: Count
`Duration`	Measures the elapsed wall clock time from when the function code starts executing as a result of an invocation to when it stops executing. This replaces the deprecated Latency metric. The maximum data point value possible is the function timeout configuration. The billed duration will be rounded up to the nearest 100 millisecond. Note that AWS Lambda only sends these metrics to CloudWatch if they have a nonzero value. Units: Milliseconds
`Throttles`	Measures the number of Lambda function invocation attempts that were throttled due to invocation rates exceeding the customer's concurrent limits (error code 429). Failed invocations may trigger a retry attempt that succeeds. Units: Count

AWS Lambda CloudWatch Dimensions

You can use the dimensions in the following table to refine the metrics returned for your Lambda functions.

Dimension	Description
`FunctionName`	Filters the data you request for a Lambda function.

Logging AWS Lambda API Calls By Using AWS CloudTrail

AWS Lambda is integrated with AWS CloudTrail, a service that captures API calls made by or on behalf of AWS Lambda in your AWS account and delivers the log files to an Amazon S3 bucket that you specify. CloudTrail captures API calls made from the AWS Lambda console or from the AWS Lambda API. Using the information collected by CloudTrail, you can determine what request was made to AWS Lambda, the source IP address from which the request was made, who made the request, when it was made, and so on. To learn more about CloudTrail, including how to configure and enable it, see the *AWS CloudTrail User Guide*.

AWS Lambda Information in CloudTrail

When CloudTrail logging is enabled in your AWS account, API calls made to AWS Lambda actions are tracked in log files. AWS Lambda records are written together with other AWS service records in a log file. CloudTrail determines when to create and write to a new file based on a time period and file size.

The following actions are supported:

- AddPermission (p. 153)
- CreateEventSourceMapping (p. 156)
- CreateFunction (p. 160)
  - The `ZipFile` parameter is omitted from the CloudTrail logs for `CreateFunction`.
- DeleteEventSourceMapping (p. 165)
- DeleteFunction (p. 167)
- GetEventSourceMapping (p. 169)
- GetFunction (p. 171)
- GetFunctionConfiguration (p. 173)
- GetPolicy (p. 176)
- ListEventSourceMappings (p. 183)
- ListFunctions (p. 185)
- RemovePermission (p. 187)
- UpdateEventSourceMapping (p. 189)

- UpdateFunctionCode (p. 192)
  - The `ZipFile` parameter is omitted from the CloudTrail logs for `UpdateFunctionCode`.
- UpdateFunctionConfiguration (p. 196)

Every log entry contains information about who generated the request. The user identity information in the log helps you determine whether the request was made with root or IAM user credentials, with temporary security credentials for a role or federated user, or by another AWS service. For more information, see the **userIdentity** field in the CloudTrail Event Reference.

You can store your log files in your bucket for as long as you want, but you can also define Amazon S3 lifecycle rules to archive or delete log files automatically. By default, your log files are encrypted by using Amazon S3 server-side encryption (SSE).

You can choose to have CloudTrail publish Amazon SNS notifications when new log files are delivered if you want to take quick action upon log file delivery. For more information, see Configuring Amazon SNS Notifications for CloudTrail.

You can also aggregate AWS Lambda log files from multiple AWS regions and multiple AWS accounts into a single S3 bucket. For more information, see Receiving CloudTrail Log Files from Multiple Sources in a Single Amazon S3 Bucket.

Understanding AWS Lambda Log File Entries

CloudTrail log files contain one or more log entries where each entry is made up of multiple JSON-formatted events. A log entry represents a single request from any source and includes information about the requested action, any parameters, the date and time of the action, and so on. The log entries are not guaranteed to be in any particular order. That is, they are not an ordered stack trace of the public API calls.

The following example shows CloudTrail log entries for the `GetFunction` and `DeleteFunction` actions.

```
{
  "Records": [
    {
      "eventVersion": "1.03",
      "userIdentity": {
        "type": "IAMUser",
        "principalId": "A1B2C3D4E5F6G7EXAMPLE",
        "arn": "arn:aws:iam::999999999999:user/myUserName",
        "accountId": "999999999999",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "userName": "myUserName"
      },
      "eventTime": "2015-03-18T19:03:36Z",
      "eventSource": "lambda.amazonaws.com",
      "eventName": "GetFunction",
      "awsRegion": "us-east-1",
      "sourceIPAddress": "127.0.0.1",
      "userAgent": "Python-httplib2/0.8 (gzip)",
      "errorCode": "AccessDenied",
      "errorMessage": "User: arn:aws:iam::999999999999:user/myUserName" is not
 authorized to perform: lambda:GetFunction on resource: arn:aws:lambda:us-west-
2:999999999999:function:other-acct-function",
      "requestParameters": null,
      "responseElements": null,
```

```
      "requestID": "7aebcd0f-cda1-11e4-aaa2-e356da31e4ff",
      "eventID": "e92a3e85-8ecd-4d23-8074-843aabfe89bf",
      "eventType": "AwsApiCall",
      "recipientAccountId": "999999999999"
    },
    {
      "eventVersion": "1.03",
      "userIdentity": {
        "type": "IAMUser",
        "principalId": "A1B2C3D4E5F6G7EXAMPLE",
        "arn": "arn:aws:iam::999999999999:user/myUserName",
        "accountId": "999999999999",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "userName": "myUserName"
      },
      "eventTime": "2015-03-18T19:04:42Z",
      "eventSource": "lambda.amazonaws.com",
      "eventName": "DeleteFunction",
      "awsRegion": "us-east-1",
      "sourceIPAddress": "127.0.0.1",
      "userAgent": "Python-httplib2/0.8 (gzip)",
      "requestParameters": {
        "functionName": "basic-node-task"
      },
      "responseElements": null,
      "requestID": "a2198ecc-cda1-11e4-aaa2-e356da31e4ff",
      "eventID": "20b84ce5-730f-482e-b2b2-e8fcc87ceb22",
      "eventType": "AwsApiCall",
      "recipientAccountId": "999999999999"
    }
  ]
}
```

Best Practices for Working with AWS Lambda Functions

The following are recommended best practices for using AWS Lambda.

- Write your Lambda function code in a stateless style, and ensure there is no affinity between your code and the underlying compute infrastructure.
- Lower costs and improve performance by minimizing the use of ‘startup’ code not directly related to processing the current event.
- Use the built-in CloudWatch monitoring of your Lambda functions to view and optimize request latencies.
- Delete old Lambda functions that you are no longer using.

AWS Lambda Limits

This section discusses AWS Lambda limits.

Topics

AWS Lambda Safety Throttles

In order to limit the impact of runaway or recursive functions during initial development and testing, AWS Lambda has a default "safety" throttle limit of 100 concurrent Lambda function executions per account. The throttle is applied to the total concurrent executions across all functions within a given region. The concurrent executions for a given function equals: (average duration of the function execution) X (number of requests or events processed by AWS Lambda). For example, if your Amazon S3 bucket sends 100 events per second to AWS Lambda, and each event takes three seconds to process, your function will handle approximately 300 concurrent executions and will exceed the default safety throttle.

> **Note**
> When used with Amazon Kinesis or with Amazon DynamoDB streams, AWS Lambda executes your function concurrently for each shard in the stream, but reads sequentially from each shard based on your batch size. For more information, see The Pull Event Model (p. 5).

If your account exceeds the safety throttle at any time, any of your functions in the region may be throttled. When your functions get throttled, you see a spike on the throttled invocations Amazon CloudWatch metric for the affected functions. If Lambda functions are invoked synchronously, it returns a throttling error (error code 429). If Lambda functions are invoked asynchronously and are throttled, they are retried for up to 15-30 minutes, which allows your function to absorb reasonable bursts of traffic. After the 15-30 minute retry period ends, incoming events are rejected as throttled.

If the Lambda function is invoked in response to Amazon S3 events, events rejected by AWS Lambda are retained and retried by Amazon S3 for up to 24 hours. Events from Amazon Kinesis streams and Amazon DynamoDB streams are retried until the Lambda function succeeds or the data expires (Amazon Kinesis and Amazon DynamoDB streams data expires after 24 hours).

To request a limit increase for concurrent execution throttle

1. Open the AWS Support Center page, sign in, if necessary, and then click **Create case**.

2. Under **Regarding**, select **Service Limit Increase**.
3. Under **Limit Type**, select **Lambda**, fill in the necessary fields in the form, and then click the button at the bottom of the page for your preferred method of contact.

Note
AWS may automatically raise the concurrent execution throttle limit on your behalf to enable your function to match the incoming event rate, as in the case of triggering the function from an Amazon S3 bucket.

List of AWS Lambda Limits

Every Lambda function is allocated with a fixed amount of specific resources regardless of the memory allocation, and each function is allocated with a fixed amount of code storage per function and per account.

The following table lists the runtime resource limits for a Lambda function per invocation.

AWS Lambda Resource Limits

Resource	Limit
Ephemeral disk capacity ("/tmp" space)	512 MB
Number of file descriptors	1,024
Number of processes and threads (combined total)	1,024
Maximum execution duration per request	60 seconds
Invoke (p. 178) request body payload size	6 MB
Invoke (p. 178) response body payload size	6 MB

The following table lists service limits for deploying a Lambda function.

AWS Lambda Deployment Limits

Item	Limit
Lambda function deployment package size (.zip/.jar file)	50 MB
Size of code/dependencies that you can zip into a deployment package (uncompressed zip/jar size)	250 MB
Total size of all the deployment packages that can be uploaded per account	1.5 GB

AWS Lambda Limit Errors

Functions that exceed any of the limits listed in the previous limits tables will fail with an `exceeded limits` exception. These limits are fixed and cannot be changed at this time. For example, if you receive the exception `CodeStorageExceededException` or an error message similar to "Code storage limit exceeded" from AWS Lambda, you need to reduce the size of your code storage.

To reduce the size of your code storage

1. Remove the functions that you no longer use.
2. Reduce the code size of the functions that you do not want to remove. You can find the code size of a Lambda function by using the AWS Lambda console, the AWS Command Line Interface, or AWS SDKs.

Appendix: API Updates

From Preview to General Availability

As part of making the service generally available, the AWS Lambda API and programming model have been updated to address customer feedback. If you are a customer who has been using Lambda before April 9, 2015, the following is a useful reference to understand what changes need to be made in order to use the new API and permissions model.

Task	Previous	New	Migration guidance for existing users
Permissions			
Setting invoke permissions for Amazon S3	Set the "invocationrole" parameter for function properties using `CreateFunction`.	Create permissions policy using `AddPermission` API for a given function. Invocationrole is deprecated.	If you use the Amazon S3 console, anytime you create a new notification rule, editing an existing rule, or deleting a rule, the console will look at all the existing rules on the same bucket and convert any existing Lambda rules as needed (remove invocation roles and automatically add permissions). If you use the Amazon S3 API for editing or updating the event source, you will need to first set permissions using Lambda `AddPermissions`, and then create the notification role.

Task	Previous	New	Migration guidance for existing users
Setting invoke permissions for Amazon Kinesis	Set the "invocationrole" parameter for function properties using `AddEventSource` API.	Add appropriate permissions to the function execution role "kinesis:GetRecords", "kinesis:GetShardIterator", "kinesis:DescribeStream", "kinesis:ListStreams"	• New event source mapping with a new function - When you create a new function, you must ensure the execution role has the necessary permissions to read from your event source. If you use the AWS Lambda console, you can use the 1-Click role creation option to automatically create a role with the necessary permissions for a given code template. You can customize the permissions on the execution role by using the IAM console (**Roles** -> Click on desired role -> scroll to inline policies -> edit policy). • New event source mapping with an existing function - Before setting up a new event source on an existing function, you need to update your execution role with stream read permissions. You can customize the permissions on the execution role by using the IAM console (**Roles** -> Click on desired role -> scroll to inline policies -> edit policy). If you are using the AWS Lambda console, you can update your execution role with the additional policy for reading and writing from Streams. • Existing event source mapping with an existing function - Existing event source mappings will continue to use the invocation role they were configured with. However, if you modify the event source through the Lambda CLI, you will need to remove the invocation role and update the execution role.
Managing functions			
Invoking a function in "Event" mode (asynchronous response)	Use `InvokeAsync` API	Use `Invoke` API with "event" `InvocationType`	Update any references to `InvokeAsync` to use `Invoke`. `InvokeAsync` is deprecated.
Creating functions programmatically	Use `UploadFunction` API	Use `CreateFunction` API	Update any references to `UploadFunction` to use `CreateFunction`
Updating function code programmatically	Use `UploadFunction` API	Use `UpdateFunctionCode` API	Update any references to `UploadFunction` to use `UpdateFunctionCode`
Managing event sources			

Task	Previous	New	Migration guidance for existing users
Creating event sources (stream to function mappings) programmatically	Use `AddEventSource` API	Use `AddEventSourceMapping`	Update any references to `AddEventSource` to use `AddEventSourceMapping` when used to create an event source
Updating event sources (stream to function mappings) programmatically	Use `AddEventSource` API	Use `UpdateEventSourceMapping`	Update any references to `AddEventSource` to use `UpdateEventSourceMapping` when used to update an event source
Retrieve single event source	`GetEventSource`	`GetEventSourceMapping`	Update any references to `GetEventSource` to use `GetEventSourceMapping` when used to update an event source
List all event sources	`ListEventSource`	`ListEventSourceMapping`	Update any references to `GetEventSource` to use `GetEventSourceMapping` when used to update an event source
Delete an event source	`RemoveEventSource`	`DeleteEventSourceMapping`	Update any references to `RemoveEventSource` to use `DeleteEventSourceMapping` when used to delete an event source
Programming model			
Specifying successful execution	`context.done(null, “successmessage”)`	`context. succeed(result)`	`context.done` will continue to work; however, it is recommended you use the new methods.
Specifying an error in execution	`context.done(errorObject, “failuremessage”)`	`context. fail(error)`	`context.done` will continue to work; however, it is recommended you use the new methods.

API Reference

This section contains the AWS Lambda API Reference documentation. When making the API calls, you will need to authenticate your request by providing a signature. AWS Lambda supports signature version 4. For more information, go to Signature Version 4 Signing Process in the *Amazon Web Services General Reference*.

For an overview of the service, see What Is AWS Lambda? (p. 1). For information about how the service works, see AWS Lambda: How it Works (p. 3).

You can use the AWS CLI to explore the AWS Lambda API. For examples, see AWS Lambda Walkthroughs (Node.js) (p. 35).

Actions

The following actions are supported:

AddPermission

Adds a permission to the access policy associated with the specified AWS Lambda function. In a "push event" model, the access policy attached to the Lambda function grants Amazon S3 or a user application permission for the Lambda `lambda:Invoke` action. For information about the push model, see AWS Lambda: How it Works. Each Lambda function has one access policy associated with it. You can use the `AddPermission` API to add a permission to the policy. You have one access policy but it can have multiple permission statements.

This operation requires permission for the `lambda:AddPermission` action.

Request Syntax

```
POST /2015-03-31/functions/FunctionName/versions/HEAD/policy HTTP/1.1
Content-type: application/json

{
    "Action": "string",
    "Principal": "string",
    "SourceAccount": "string",
    "SourceArn": "string",
    "StatementId": "string"
}
```

URI Request Parameters

The request requires the following URI parameters.

FunctionName
Name of the Lambda function whose access policy you are updating by adding a new permission.

You can specify an unqualified function name (for example, "Thumbnail") or you can specify Amazon Resource Name (ARN) of the function (for example, "arn:aws:lambda:us-west-2:account-id:function:ThumbNail"). AWS Lambda also allows you to specify only the account ID qualifier (for example, "account-id:Thumbnail"). Note that the length constraint applies only to the ARN. If you specify only the function name, it is limited to 64 character in length.

Length constraints: Minimum length of 1. Maximum length of 111.

Pattern:
`(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)`

Request Body

The request requires the following data in JSON format.

Action
The AWS Lambda action you want to allow in this statement. Each Lambda action is a string starting with "lambda:" followed by the API name (see Actions (p. 152)). For example, "lambda:CreateFunction". You can use wildcard ("lambda:*") to grant permission for all AWS Lambda actions.

Type: String

Pattern: `(lambda:[*]|lambda:[a-zA-Z]+|[*])`

Required: Yes

Principal

The principal who is getting this permission. It can be Amazon S3 service Principal ("s3.amazonaws.com") if you want Amazon S3 to invoke the function, an AWS account ID if you are granting cross-account permission, or any valid AWS service principal such as "sns.amazonaws.com". For example, you might want to allow a custom application in another AWS account to push events to AWS Lambda by invoking your function.

Type: String

Pattern: `.*`

Required: Yes

SourceAccount

The AWS account ID (without a hyphen) of the source owner. For example, if the `SourceArn` identifies a bucket, then this is the bucket owner's account ID. You can use this additional condition to ensure the bucket you specify is owned by a specific account (it is possible the bucket owner deleted the bucket and some other AWS account created the bucket). You can also use this condition to specify all sources (that is, you don't specify the `SourceArn`) owned by a specific account.

Type: String

Pattern: `\d{12}`

Required: No

SourceArn

This is optional; however, when granting Amazon S3 permission to invoke your function, you should specify this field with the bucket Amazon Resource Name (ARN) as its value. This ensures that only events generated from the specified bucket can invoke the function.

> **Important**
> If you add a permission for the Amazon S3 principal without providing the source ARN, any AWS account that creates a mapping to your function ARN can send events to invoke your Lambda function from Amazon S3.

Type: String

Pattern: `arn:aws:([a-zA-Z0-9\-])+:([a-z]{2}-[a-z]+-\d{1})?:(\d{12})?:(.*)`

Required: No

StatementId

A unique statement identifier.

Type: String

Length constraints: Minimum length of 1. Maximum length of 100.

Pattern: `([a-zA-Z0-9-_]+)`

Required: Yes

Response Syntax

```
HTTP/1.1 201
Content-type: application/json

{
```

```
    "Statement": "string"
}
```

Response Elements

If the action is successful, the service sends back an HTTP 201 response.

The following data is returned in JSON format by the service.

Statement
The permission statement you specified in the request. The response returns the same as a string using "\" as an escape character in the JSON.

Type: String

Errors

InvalidParameterValueException
One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

PolicyLengthExceededException
Lambda function access policy is limited to 20 KB.

HTTP Status Code: 400

ResourceConflictException
The resource already exists.

HTTP Status Code: 400

ResourceNotFoundException
The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 400

ServiceException
The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException
HTTP Status Code: 400

CreateEventSourceMapping

Identifies a stream as an event source for a Lambda function. It can be either an Amazon Kinesis stream or an Amazon DynamoDB stream. AWS Lambda invokes the specified function when records are posted to the stream.

This is the pull model, where AWS Lambda invokes the function. For more information, go to AWS Lambda: How it Works in the *AWS Lambda Developer Guide*.

This association between an Amazon Kinesis stream and a Lambda function is called the event source mapping. You provide the configuration information (for example, which stream to read from and which Lambda function to invoke) for the event source mapping in the request body.

Each event source, such as an Amazon Kinesis or a DynamoDB stream, can be associated with multiple AWS Lambda function. A given Lambda function can be associated with multiple AWS event sources.

This operation requires permission for the `lambda:CreateEventSourceMapping` action.

Request Syntax

```
POST /2015-03-31/event-source-mappings/ HTTP/1.1
Content-type: application/json

{
    "BatchSize": number,
    "Enabled": boolean,
    "EventSourceArn": "string",
    "FunctionName": "string",
    "StartingPosition": "string"
}
```

URI Request Parameters

The request does not use any URI parameters.

Request Body

The request requires the following data in JSON format.

BatchSize

The largest number of records that AWS Lambda will retrieve from your event source at the time of invoking your function. Your function receives an event with all the retrieved records. The default is 100 records.

Type: Number

Valid range: Minimum value of 1. Maximum value of 10000.

Required: No

Enabled

Indicates whether AWS Lambda should begin polling the event source, the default is not enabled.

Type: Boolean

Required: No

EventSourceArn

The Amazon Resource Name (ARN) of the Amazon Kinesis or the Amazon DynamoDB stream that is the event source. Any record added to this stream could cause AWS Lambda to invoke your Lambda function, it depends on the `BatchSize`. AWS Lambda POSTs the Amazon Kinesis event, containing records, to your Lambda function as JSON.

Type: String

Pattern: `arn:aws:([a-zA-Z0-9\-])+:([a-z]{2}-[a-z]+-\d{1})?:(\d{12})?:(.*)`

Required: Yes

FunctionName

The Lambda function to invoke when AWS Lambda detects an event on the stream.

You can specify an unqualified function name (for example, "Thumbnail") or you can specify Amazon Resource Name (ARN) of the function (for example, "arn:aws:lambda:us-west-2:account-id:function:ThumbNail"). AWS Lambda also allows you to specify only the account ID qualifier (for example, "account-id:Thumbnail"). Note that the length constraint applies only to the ARN. If you specify only the function name, it is limited to 64 character in length.

Type: String

Length constraints: Minimum length of 1. Maximum length of 111.

Pattern:
`(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)`

Required: Yes

StartingPosition

The position in the stream where AWS Lambda should start reading. For more information, go to ShardIteratorType in the *Amazon Kinesis API Reference*.

Type: String

Valid Values: `TRIM_HORIZON | LATEST`

Required: Yes

Response Syntax

```
HTTP/1.1 202
Content-type: application/json

{
    "BatchSize": number,
    "EventSourceArn": "string",
    "FunctionArn": "string",
    "LastModified": number,
    "LastProcessingResult": "string",
    "State": "string",
    "StateTransitionReason": "string",
    "UUID": "string"
}
```

Response Elements

If the action is successful, the service sends back an HTTP 202 response.

The following data is returned in JSON format by the service.

BatchSize
The largest number of records that AWS Lambda will retrieve from your event source at the time of invoking your function. Your function receives an event with all the retrieved records.

Type: Number

Valid range: Minimum value of 1. Maximum value of 10000.

EventSourceArn
The Amazon Resource Name (ARN) of the Amazon Kinesis stream that is the source of events.

Type: String

Pattern: `arn:aws:([a-zA-Z0-9\-])+:([a-z]{2}-[a-z]+-\d{1})?:(\d{12})?:(.*)`

FunctionArn
The Lambda function to invoke when AWS Lambda detects an event on the stream.

Type: String

Pattern: arn:aws:lambda:[a-z]{2}-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(\/[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12})?

LastModified
The UTC time string indicating the last time the event mapping was updated.

Type: DateTime

LastProcessingResult
The result of the last AWS Lambda invocation of your Lambda function.

Type: String

State
The state of the event source mapping. It can be "Creating", "Enabled", "Disabled", "Enabling", "Disabling", "Updating", or "Deleting".

Type: String

StateTransitionReason
The reason the event source mapping is in its current state. It is either user-requested or an AWS Lambda-initiated state transition.

Type: String

UUID
The AWS Lambda assigned opaque identifier for the mapping.

Type: String

Errors

InvalidParameterValueException
One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

ResourceConflictException
The resource already exists.

HTTP Status Code: 400

ServiceException
The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException
HTTP Status Code: 400

CreateFunction

Creates a new Lambda function. The function metadata is created from the request parameters, and the code for the function is provided by a .zip file in the request body. If the function name already exists, the operation will fail. Note that the function name is case-sensitive.

This operation requires permission for the `lambda:CreateFunction` action.

Request Syntax

```
POST /2015-03-31/functions HTTP/1.1
Content-type: application/json

{
    "Code": {
        "S3Bucket": "string",
        "S3Key": "string",
        "S3ObjectVersion": "string",
        "ZipFile": blob
    },
    "Description": "string",
    "FunctionName": "string",
    "Handler": "string",
    "MemorySize": number,
    "Role": "string",
    "Runtime": "string",
    "Timeout": number
}
```

URI Request Parameters

The request does not use any URI parameters.

Request Body

The request requires the following data in JSON format.

Code
The code for the Lambda function.

Type: FunctionCode (p. 201) object

Required: Yes

Description
A short, user-defined function description. Lambda does not use this value. Assign a meaningful description as you see fit.

Type: String

Length constraints: Minimum length of 0. Maximum length of 256.

Required: No

FunctionName
The name you want to assign to the function you are uploading. You can specify an unqualified function name (for example, "Thumbnail") or you can specify Amazon Resource Name (ARN) of the function (for example, "arn:aws:lambda:us-west-2:account-id:function:ThumbNail"). AWS Lambda

also allows you to specify only the account ID qualifier (for example, "account-id:Thumbnail"). Note that the length constraint applies only to the ARN. If you specify only the function name, it is limited to 64 character in length. The function names appear in the console and are returned in the ListFunctions (p. 185) API. Function names are used to specify functions to other AWS Lambda APIs, such as Invoke (p. 178).

Type: String

Length constraints: Minimum length of 1. Maximum length of 111.

Pattern:
`(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)`

Required: Yes

Handler

The function within your code that Lambda calls to begin execution. For Node.js, it is the *module-name.export* value in your function. For Java, it can be `package.class-name::handler` or `package.class-name`. For more information, see Lambda Function Handler (Java).

Type: String

Length constraints: Minimum length of 0. Maximum length of 128.

Pattern: `[^\s]+`

Required: Yes

MemorySize

The amount of memory, in MB, your Lambda function is given. Lambda uses this memory size to infer the amount of CPU and memory allocated to your function. Your function use-case determines your CPU and memory requirements. For example, a database operation might need less memory compared to an image processing function. The default value is 128 MB. The value must be a multiple of 64 MB.

Type: Number

Valid range: Minimum value of 128. Maximum value of 1536.

Required: No

Role

The Amazon Resource Name (ARN) of the IAM role that Lambda assumes when it executes your function to access any other Amazon Web Services (AWS) resources. For more information, see AWS Lambda: How it Works

Type: String

Pattern: `arn:aws:iam::\d{12}:role/?[a-zA-Z_0-9+=,.@\-_/]+`

Required: Yes

Runtime

The runtime environment for the Lambda function you are uploading. Currently, Lambda supports "java" and "nodejs" as the runtime.

Type: String

Valid Values: `nodejs | java8`

Required: Yes

Timeout
The function execution time at which Lambda should terminate the function. Because the execution time has cost implications, we recommend you set this value based on your expected execution time. The default is 3 seconds.

Type: Number

Valid range: Minimum value of 1. Maximum value of 60.

Required: No

Response Syntax

```
HTTP/1.1 201
Content-type: application/json

{
    "CodeSize": number,
    "Description": "string",
    "FunctionArn": "string",
    "FunctionName": "string",
    "Handler": "string",
    "LastModified": "string",
    "MemorySize": number,
    "Role": "string",
    "Runtime": "string",
    "Timeout": number
}
```

Response Elements

If the action is successful, the service sends back an HTTP 201 response.

The following data is returned in JSON format by the service.

CodeSize
The size, in bytes, of the function .zip file you uploaded.

Type: Long

Description
The user-provided description.

Type: String

Length constraints: Minimum length of 0. Maximum length of 256.

FunctionArn
The Amazon Resource Name (ARN) assigned to the function.

Type: String

Pattern:
arn:aws:lambda:[a-z]{2}-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(\/[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12})?

FunctionName
The name of the function.

Type: String

Length constraints: Minimum length of 1. Maximum length of 111.

Pattern: `(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)`

Handler
The function Lambda calls to begin executing your function.

Type: String

Length constraints: Minimum length of 0. Maximum length of 128.

Pattern: `[^\s]+`

LastModified
The timestamp of the last time you updated the function.

Type: String

MemorySize
The memory size, in MB, you configured for the function. Must be a multiple of 64 MB.

Type: Number

Valid range: Minimum value of 128. Maximum value of 1536.

Role
The Amazon Resource Name (ARN) of the IAM role that Lambda assumes when it executes your function to access any other Amazon Web Services (AWS) resources.

Type: String

Pattern: `arn:aws:iam::\d{12}:role/?[a-zA-Z_0-9+=,.@\-_/]+`

Runtime
The runtime environment for the Lambda function.

Type: String

Valid Values: `nodejs | java8`

Timeout
The function execution time at which Lambda should terminate the function. Because the execution time has cost implications, we recommend you set this value based on your expected execution time. The default is 3 seconds.

Type: Number

Valid range: Minimum value of 1. Maximum value of 60.

Errors

CodeStorageExceededException
HTTP Status Code: 400

InvalidParameterValueException
One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

ResourceConflictException
The resource already exists.

HTTP Status Code: 400

ResourceNotFoundException
The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 400

ServiceException
The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException
HTTP Status Code: 400

DeleteEventSourceMapping

Removes an event source mapping. This means AWS Lambda will no longer invoke the function for events in the associated source.

This operation requires permission for the `lambda:DeleteEventSourceMapping` action.

Request Syntax

```
DELETE /2015-03-31/event-source-mappings/UUID HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

UUID
The event source mapping ID.

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 202
Content-type: application/json

{
    "BatchSize": number,
    "EventSourceArn": "string",
    "FunctionArn": "string",
    "LastModified": number,
    "LastProcessingResult": "string",
    "State": "string",
    "StateTransitionReason": "string",
    "UUID": "string"
}
```

Response Elements

If the action is successful, the service sends back an HTTP 202 response.

The following data is returned in JSON format by the service.

BatchSize
The largest number of records that AWS Lambda will retrieve from your event source at the time of invoking your function. Your function receives an event with all the retrieved records.

Type: Number

Valid range: Minimum value of 1. Maximum value of 10000.

EventSourceArn
The Amazon Resource Name (ARN) of the Amazon Kinesis stream that is the source of events.

Type: String

Pattern: `arn:aws:([a-zA-Z0-9\-])+:([a-z]{2}-[a-z]+-\d{1})?:(\d{12})?:(.*)`

FunctionArn
The Lambda function to invoke when AWS Lambda detects an event on the stream.

Type: String

Pattern: `arn:aws:lambda:[a-z]{2}-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(\/[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12})?`

LastModified
The UTC time string indicating the last time the event mapping was updated.

Type: DateTime

LastProcessingResult
The result of the last AWS Lambda invocation of your Lambda function.

Type: String

State
The state of the event source mapping. It can be "Creating", "Enabled", "Disabled", "Enabling", "Disabling", "Updating", or "Deleting".

Type: String

StateTransitionReason
The reason the event source mapping is in its current state. It is either user-requested or an AWS Lambda-initiated state transition.

Type: String

UUID
The AWS Lambda assigned opaque identifier for the mapping.

Type: String

Errors

InvalidParameterValueException
One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

ResourceNotFoundException
The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 400

ServiceException
The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException
HTTP Status Code: 400

DeleteFunction

Deletes the specified Lambda function code and configuration.

When you delete a function the associated access policy is also deleted. You will need to delete the event source mappings explicitly.

This operation requires permission for the `lambda:DeleteFunction` action.

Request Syntax

```
DELETE /2015-03-31/functions/FunctionName HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

FunctionName
The Lambda function to delete.

You can specify an unqualified function name (for example, "Thumbnail") or you can specify Amazon Resource Name (ARN) of the function (for example, "arn:aws:lambda:us-west-2:account-id:function:ThumbNail"). AWS Lambda also allows you to specify only the account ID qualifier (for example, "account-id:Thumbnail"). Note that the length constraint applies only to the ARN. If you specify only the function name, it is limited to 64 character in length.

Length constraints: Minimum length of 1. Maximum length of 111.

Pattern:
`(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)`

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 204
```

Response Elements

If the action is successful, the service sends back an HTTP 204 response with an empty HTTP body.

Errors

ResourceNotFoundException
The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 400

ServiceException
The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException

HTTP Status Code: 400

GetEventSourceMapping

Returns configuration information for the specified event source mapping (see CreateEventSourceMapping (p. 156)).

This operation requires permission for the `lambda:GetEventSourceMapping` action.

Request Syntax

```
GET /2015-03-31/event-source-mappings/UUID HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

UUID
The AWS Lambda assigned ID of the event source mapping.

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "BatchSize": number,
    "EventSourceArn": "string",
    "FunctionArn": "string",
    "LastModified": number,
    "LastProcessingResult": "string",
    "State": "string",
    "StateTransitionReason": "string",
    "UUID": "string"
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

BatchSize
The largest number of records that AWS Lambda will retrieve from your event source at the time of invoking your function. Your function receives an event with all the retrieved records.

Type: Number

Valid range: Minimum value of 1. Maximum value of 10000.

EventSourceArn
The Amazon Resource Name (ARN) of the Amazon Kinesis stream that is the source of events.

Type: String

Pattern: `arn:aws:([a-zA-Z0-9\-])+:([a-z]{2}-[a-z]+-\d{1})?:(\d{12})?:(.*)`

FunctionArn
The Lambda function to invoke when AWS Lambda detects an event on the stream.

Type: String

Pattern: arn:aws:lambda:[a-z]{2}-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(\/[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12})?

LastModified
The UTC time string indicating the last time the event mapping was updated.

Type: DateTime

LastProcessingResult
The result of the last AWS Lambda invocation of your Lambda function.

Type: String

State
The state of the event source mapping. It can be "Creating", "Enabled", "Disabled", "Enabling", "Disabling", "Updating", or "Deleting".

Type: String

StateTransitionReason
The reason the event source mapping is in its current state. It is either user-requested or an AWS Lambda-initiated state transition.

Type: String

UUID
The AWS Lambda assigned opaque identifier for the mapping.

Type: String

Errors

InvalidParameterValueException
One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

ResourceNotFoundException
The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 400

ServiceException
The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException
HTTP Status Code: 400

GetFunction

Returns the configuration information of the Lambda function and a presigned URL link to the .zip file you uploaded with CreateFunction (p. 160) so you can download the .zip file. Note that the URL is valid for up to 10 minutes. The configuration information is the same information you provided as parameters when uploading the function.

This operation requires permission for the `lambda:GetFunction` action.

Request Syntax

```
GET /2015-03-31/functions/FunctionName/versions/HEAD HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

FunctionName
The Lambda function name.

You can specify an unqualified function name (for example, "Thumbnail") or you can specify Amazon Resource Name (ARN) of the function (for example, "arn:aws:lambda:us-west-2:account-id:function:ThumbNail"). AWS Lambda also allows you to specify only the account ID qualifier (for example, "account-id:Thumbnail"). Note that the length constraint applies only to the ARN. If you specify only the function name, it is limited to 64 character in length.

Length constraints: Minimum length of 1. Maximum length of 111.

Pattern:
`(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)`

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "Code": {
        "Location": "string",
        "RepositoryType": "string"
    },
    "Configuration": {
        "CodeSize": number,
        "Description": "string",
        "FunctionArn": "string",
        "FunctionName": "string",
        "Handler": "string",
        "LastModified": "string",
        "MemorySize": number,
        "Role": "string",
```

```
        "Runtime": "string",
        "Timeout": number
    }
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

Code
The object for the Lambda function location.

Type: FunctionCodeLocation (p. 202) object

Configuration
A complex type that describes function metadata.

Type: FunctionConfiguration (p. 202) object

Errors

ResourceNotFoundException
The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 400

ServiceException
The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException
HTTP Status Code: 400

GetFunctionConfiguration

Returns the configuration information of the Lambda function. This the same information you provided as parameters when uploading the function by using CreateFunction (p. 160).

This operation requires permission for the `lambda:GetFunctionConfiguration` operation.

Request Syntax

```
GET /2015-03-31/functions/FunctionName/versions/HEAD/configuration HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

FunctionName

The name of the Lambda function for which you want to retrieve the configuration information.

You can specify an unqualified function name (for example, "Thumbnail") or you can specify Amazon Resource Name (ARN) of the function (for example, "arn:aws:lambda:us-west-2:account-id:function:ThumbNail"). AWS Lambda also allows you to specify only the account ID qualifier (for example, "account-id:Thumbnail"). Note that the length constraint applies only to the ARN. If you specify only the function name, it is limited to 64 character in length.

Length constraints: Minimum length of 1. Maximum length of 111.

Pattern: `(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)`

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "CodeSize": number,
    "Description": "string",
    "FunctionArn": "string",
    "FunctionName": "string",
    "Handler": "string",
    "LastModified": "string",
    "MemorySize": number,
    "Role": "string",
    "Runtime": "string",
    "Timeout": number
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

CodeSize
The size, in bytes, of the function .zip file you uploaded.

Type: Long

Description
The user-provided description.

Type: String

Length constraints: Minimum length of 0. Maximum length of 256.

FunctionArn
The Amazon Resource Name (ARN) assigned to the function.

Type: String

Pattern: `arn:aws:lambda:[a-z]{2}-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(\:[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12})?`

FunctionName
The name of the function.

Type: String

Length constraints: Minimum length of 1. Maximum length of 111.

Pattern: `(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)`

Handler
The function Lambda calls to begin executing your function.

Type: String

Length constraints: Minimum length of 0. Maximum length of 128.

Pattern: `[^\s]+`

LastModified
The timestamp of the last time you updated the function.

Type: String

MemorySize
The memory size, in MB, you configured for the function. Must be a multiple of 64 MB.

Type: Number

Valid range: Minimum value of 128. Maximum value of 1536.

Role
The Amazon Resource Name (ARN) of the IAM role that Lambda assumes when it executes your function to access any other Amazon Web Services (AWS) resources.

Type: String

Pattern: `arn:aws:iam::\d{12}:role/?[a-zA-Z_0-9+=,.@\-_/]+`

Runtime
The runtime environment for the Lambda function.

Type: String

Valid Values: `nodejs | java8`

Timeout
The function execution time at which Lambda should terminate the function. Because the execution time has cost implications, we recommend you set this value based on your expected execution time. The default is 3 seconds.

Type: Number

Valid range: Minimum value of 1. Maximum value of 60.

Errors

ResourceNotFoundException
The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 400

ServiceException
The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException
HTTP Status Code: 400

GetPolicy

Returns the access policy, containing a list of permissions granted via the `AddPermission` API, associated with the specified bucket.

You need permission for the `lambda:GetPolicy action.`

Request Syntax

```
GET /2015-03-31/functions/FunctionName/versions/HEAD/policy HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

FunctionName
Function name whose access policy you want to retrieve.

You can specify an unqualified function name (for example, "Thumbnail") or you can specify Amazon Resource Name (ARN) of the function (for example, "arn:aws:lambda:us-west-2:account-id:function:ThumbNail"). AWS Lambda also allows you to specify only the account ID qualifier (for example, "account-id:Thumbnail"). Note that the length constraint applies only to the ARN. If you specify only the function name, it is limited to 64 character in length.

Length constraints: Minimum length of 1. Maximum length of 111.

Pattern:
`(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)`

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "Policy": "string"
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

Policy
The access policy associated with the specified function. The response returns the same as a string using "\" as an escape character in the JSON.

Type: String

Errors

ResourceNotFoundException
The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 400

ServiceException
The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException
HTTP Status Code: 400

Invoke

Invokes a specified Lambda function.

This operation requires permission for the `lambda:InvokeFunction` action.

Request Syntax

```
POST /2015-03-31/functions/FunctionName/invocations HTTP/1.1
X-Amz-Client-Context: ClientContext
X-Amz-Invocation-Type: InvocationType
X-Amz-Log-Type: LogType

Payload
```

URI Request Parameters

The request requires the following URI parameters.

ClientContext
Using the `ClientContext` you can pass client-specific information to the Lambda function you are invoking. You can then process the client information in your Lambda function as you choose through the context variable. For an example of a ClientContext JSON, go to PutEvents in the *Amazon Mobile Analytics API Reference and User Guide*.

The ClientContext JSON must be base64-encoded.

FunctionName
The Lambda function name.

You can specify an unqualified function name (for example, "Thumbnail") or you can specify Amazon Resource Name (ARN) of the function (for example, "arn:aws:lambda:us-west-2:account-id:function:ThumbNail"). AWS Lambda also allows you to specify only the account ID qualifier (for example, "account-id:Thumbnail"). Note that the length constraint applies only to the ARN. If you specify only the function name, it is limited to 64 character in length.

Length constraints: Minimum length of 1. Maximum length of 111.

Pattern:
`(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)`

InvocationType
By default, the `Invoke` API assumes "RequestResponse" invocation type. You can optionally request asynchronous execution by specifying "Event" as the `InvocationType`. You can also use this parameter to request AWS Lambda to not execute the function but do some verification, such as if the caller is authorized to invoke the function and if the inputs are valid. You request this by specifying "DryRun" as the `InvocationType`. This is useful in a cross-account scenario when you want to verify access to a function without running it.

Valid Values: `Event | RequestResponse | DryRun`

LogType
You can set this optional parameter to "Tail" in the request only if you specify the `InvocationType` parameter with value "RequestResponse". In this case, AWS Lambda returns the base64-encoded last 4 KB of log data produced by your Lambda function in the `x-amz-log-results` header.

Valid Values: `None | Tail`

Request Body

The request requires the following as the HTTP body.

Payload
JSON that you want to provide to your Lambda function as input.

Response Syntax

```
HTTP/1.1 StatusCode
X-Amz-Function-Error: FunctionError
X-Amz-Log-Result: LogResult

Payload
```

Response Elements

StatusCode
The HTTP status code will be in the 200 range for successful request. For the "RequestResonse" invocation type this status code will be 200. For the "Event" invocation type this status code will be 202. For the "DryRun" invocation type the status code will be 204.

The response returns the following HTTP headers.

FunctionError
Indicates whether an error occurred while executing the Lambda function. If an error occurred this field will have one of two values; `Handled` or `Unhandled`. `Handled` errors are errors that are reported by the function while the `Unhandled` errors are those detected and reported by AWS Lambda. Unhandled errors include out of memory errors and function timeouts. For information about how to report an `Handled` error, see Programming Model.

LogResult
It is the base64-encoded logs for the Lambda function invocation. This is present only if the invocation type is "RequestResponse" and the logs were requested.

The response returns the following as the HTTP body.

Payload
It is the JSON representation of the object returned by the Lambda function. In This is present only if the invocation type is "RequestResponse".

In the event of a function error this field contains a message describing the error. For the `Handled` errors the Lambda function will report this message. For `Unhandled` errors AWS Lambda reports the message.

Errors

InvalidRequestContentException
The request body could not be parsed as JSON.

HTTP Status Code: 400

RequestTooLargeException
HTTP Status Code: 400

ResourceNotFoundException
The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 400

ServiceException
The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException
HTTP Status Code: 400

UnsupportedMediaTypeException
HTTP Status Code: 400

InvokeAsync

Important
This API is deprecated. We recommend you use `Invoke` API (see Invoke (p. 178)).

Submits an invocation request to AWS Lambda. Upon receiving the request, Lambda executes the specified function asynchronously. To see the logs generated by the Lambda function execution, see the CloudWatch logs console.

This operation requires permission for the `lambda:InvokeFunction` action.

Request Syntax

```
POST /2014-11-13/functions/FunctionName/invoke-async/ HTTP/1.1

InvokeArgs
```

URI Request Parameters

The request requires the following URI parameters.

FunctionName
The Lambda function name.

Length constraints: Minimum length of 1. Maximum length of 111.

Pattern:
`(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)`

Request Body

The request requires the following as the HTTP body.

InvokeArgs
JSON that you want to provide to your Lambda function as input.

Response Syntax

```
HTTP/1.1 Status
```

Response Elements

Status
It will be 202 upon success.

Errors

InvalidRequestContentException
The request body could not be parsed as JSON.

HTTP Status Code: 400

ResourceNotFoundException
The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 400

ServiceException
The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

Examples

Invoke a Lambda function

The following example uses a POST request to invoke a Lambda function.

Sample Request

```
POST /2014-11-13/functions/helloworld/invoke-async/ HTTP/1.1
[input json]
```

Sample Response

```
HTTP/1.1 202 Accepted

x-amzn-requestid: f037bc5c-5a08-11e4-b02e-af446c3f9d0d
content-length: 0
connection: keep-alive
date: Wed, 22 Oct 2014 16:31:55 GMT
content-type: application/json
```

ListEventSourceMappings

Returns a list of event source mappings you created using the `CreateEventSourceMapping` (see CreateEventSourceMapping (p. 156)), where you identify a stream as an event source. This list does not include Amazon S3 event sources.

For each mapping, the API returns configuration information. You can optionally specify filters to retrieve specific event source mappings.

This operation requires permission for the `lambda:ListEventSourceMappings` action.

Request Syntax

```
GET /2015-03-31/event-source-mappings/?Marker=Marker&MaxItems=MaxItems&Function
Name=FunctionName&EventSourceArn=EventSourceArn HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

EventSourceArn
The Amazon Resource Name (ARN) of the Amazon Kinesis stream.

Pattern: `arn:aws:([a-zA-Z0-9\-])+:([a-z]{2}-[a-z]+-\d{1})?:(\d{12})?:(.*)`

FunctionName
The name of the Lambda function.

You can specify an unqualified function name (for example, "Thumbnail") or you can specify Amazon Resource Name (ARN) of the function (for example, "arn:aws:lambda:us-west-2:account-id:function:ThumbNail"). AWS Lambda also allows you to specify only the account ID qualifier (for example, "account-id:Thumbnail"). Note that the length constraint applies only to the ARN. If you specify only the function name, it is limited to 64 character in length.

Length constraints: Minimum length of 1. Maximum length of 111.

Pattern:
`(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)`

Marker
Optional string. An opaque pagination token returned from a previous `ListEventSourceMappings` operation. If present, specifies to continue the list from where the returning call left off.

MaxItems
Optional integer. Specifies the maximum number of event sources to return in response. This value must be greater than 0.

Valid range: Minimum value of 1. Maximum value of 10000.

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "EventSourceMappings": [
        {
            "BatchSize": number,
            "EventSourceArn": "string",
            "FunctionArn": "string",
            "LastModified": number,
            "LastProcessingResult": "string",
            "State": "string",
            "StateTransitionReason": "string",
            "UUID": "string"
        }
    ],
    "NextMarker": "string"
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

EventSourceMappings
An array of `EventSourceMappingConfiguration` objects.

Type: array of EventSourceMappingConfiguration (p. 200) objects

NextMarker
A string, present if there are more event source mappings.

Type: String

Errors

InvalidParameterValueException
One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

ResourceNotFoundException
The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 400

ServiceException
The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException
HTTP Status Code: 400

ListFunctions

Returns a list of your Lambda functions. For each function, the response includes the function configuration information. You must use GetFunction (p. 171) to retrieve the code for your function.

This operation requires permission for the `lambda:ListFunctions` action.

Request Syntax

```
GET /2015-03-31/functions/?Marker=Marker&MaxItems=MaxItems HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

Marker
Optional string. An opaque pagination token returned from a previous `ListFunctions` operation. If present, indicates where to continue the listing.

MaxItems
Optional integer. Specifies the maximum number of AWS Lambda functions to return in response. This parameter value must be greater than 0.

Valid range: Minimum value of 1. Maximum value of 10000.

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "Functions": [
        {
            "CodeSize": number,
            "Description": "string",
            "FunctionArn": "string",
            "FunctionName": "string",
            "Handler": "string",
            "LastModified": "string",
            "MemorySize": number,
            "Role": "string",
            "Runtime": "string",
            "Timeout": number
        }
    ],
    "NextMarker": "string"
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

Functions
A list of Lambda functions.

Type: array of FunctionConfiguration (p. 202) objects

NextMarker
A string, present if there are more functions.

Type: String

Errors

ServiceException
The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException
HTTP Status Code: 400

RemovePermission

You can remove individual permissions from an access policy associated with a Lambda function by providing a Statement ID.

Note that removal of a permission will cause an active event source to lose permission to the function.

You need permission for the `lambda:RemovePermission` action.

Request Syntax

```
DELETE /2015-03-31/functions/FunctionName/versions/HEAD/policy/StatementId HTTP/1.1
```

URI Request Parameters

The request requires the following URI parameters.

FunctionName

Lambda function whose access policy you want to remove a permission from.

You can specify an unqualified function name (for example, "Thumbnail") or you can specify Amazon Resource Name (ARN) of the function (for example, "arn:aws:lambda:us-west-2:account-id:function:ThumbNail"). AWS Lambda also allows you to specify only the account ID qualifier (for example, "account-id:Thumbnail"). Note that the length constraint applies only to the ARN. If you specify only the function name, it is limited to 64 character in length.

Length constraints: Minimum length of 1. Maximum length of 111.

Pattern: `(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)`

StatementId

Statement ID of the permission to remove.

Length constraints: Minimum length of 1. Maximum length of 100.

Pattern: `([a-zA-Z0-9-_]+)`

Request Body

The request does not have a request body.

Response Syntax

```
HTTP/1.1 204
```

Response Elements

If the action is successful, the service sends back an HTTP 204 response with an empty HTTP body.

Errors

ResourceNotFoundException
The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 400

ServiceException
The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException
HTTP Status Code: 400

UpdateEventSourceMapping

You can update an event source mapping. This is useful if you want to change the parameters of the existing mapping without losing your position in the stream. You can change which function will receive the stream records, but to change the stream itself, you must create a new mapping.

This operation requires permission for the `lambda:UpdateEventSourceMapping` action.

Request Syntax

```
PUT /2015-03-31/event-source-mappings/UUID HTTP/1.1
Content-type: application/json

{
    "BatchSize": number,
    "Enabled": boolean,
    "FunctionName": "string"
}
```

URI Request Parameters

The request requires the following URI parameters.

UUID
The event source mapping identifier.

Request Body

The request requires the following data in JSON format.

BatchSize
The maximum number of stream records that can be sent to your Lambda function for a single invocation.

Type: Number

Valid range: Minimum value of 1. Maximum value of 10000.

Required: No

Enabled
Specifies whether AWS Lambda should actively poll the stream or not. If disabled, AWS Lambda will not poll the stream.

Type: Boolean

Required: No

FunctionName
The Lambda function to which you want the stream records sent.

You can specify an unqualified function name (for example, "Thumbnail") or you can specify Amazon Resource Name (ARN) of the function (for example, "arn:aws:lambda:us-west-2:account-id:function:ThumbNail"). AWS Lambda also allows you to specify only the account ID qualifier (for example, "account-id:Thumbnail"). Note that the length constraint applies only to the ARN. If you specify only the function name, it is limited to 64 character in length.

Type: String

Length constraints: Minimum length of 1. Maximum length of 111.

Pattern:
`(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)`

Required: No

Response Syntax

```
HTTP/1.1 202
Content-type: application/json

{
    "BatchSize": number,
    "EventSourceArn": "string",
    "FunctionArn": "string",
    "LastModified": number,
    "LastProcessingResult": "string",
    "State": "string",
    "StateTransitionReason": "string",
    "UUID": "string"
}
```

Response Elements

If the action is successful, the service sends back an HTTP 202 response.

The following data is returned in JSON format by the service.

BatchSize
The largest number of records that AWS Lambda will retrieve from your event source at the time of invoking your function. Your function receives an event with all the retrieved records.

Type: Number

Valid range: Minimum value of 1. Maximum value of 10000.

EventSourceArn
The Amazon Resource Name (ARN) of the Amazon Kinesis stream that is the source of events.

Type: String

Pattern: `arn:aws:([a-zA-Z0-9\-])+:([a-z]{2}-[a-z]+-\d{1})?:(\d{12})?:(.*)`

FunctionArn
The Lambda function to invoke when AWS Lambda detects an event on the stream.

Type: String

Pattern:
`arn:aws:lambda:[a-z]{2}-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(\/[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12})?`

LastModified
The UTC time string indicating the last time the event mapping was updated.

Type: DateTime

LastProcessingResult
The result of the last AWS Lambda invocation of your Lambda function.

Type: String

State
The state of the event source mapping. It can be "Creating", "Enabled", "Disabled", "Enabling", "Disabling", "Updating", or "Deleting".

Type: String

StateTransitionReason
The reason the event source mapping is in its current state. It is either user-requested or an AWS Lambda-initiated state transition.

Type: String

UUID
The AWS Lambda assigned opaque identifier for the mapping.

Type: String

Errors

InvalidParameterValueException
One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

ResourceNotFoundException
The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 400

ServiceException
The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException
HTTP Status Code: 400

UpdateFunctionCode

Updates the code for the specified Lambda function. This operation must only be used on an existing Lambda function and cannot be used to update the function configuration.

This operation requires permission for the `lambda:UpdateFunctionCode` action.

Request Syntax

```
PUT /2015-03-31/functions/FunctionName/versions/HEAD/code HTTP/1.1
Content-type: application/json

{
    "S3Bucket": "string",
    "S3Key": "string",
    "S3ObjectVersion": "string",
    "ZipFile": blob
}
```

URI Request Parameters

The request requires the following URI parameters.

FunctionName
The existing Lambda function name whose code you want to replace.

You can specify an unqualified function name (for example, "Thumbnail") or you can specify Amazon Resource Name (ARN) of the function (for example, "arn:aws:lambda:us-west-2:account-id:function:ThumbNail"). AWS Lambda also allows you to specify only the account ID qualifier (for example, "account-id:Thumbnail"). Note that the length constraint applies only to the ARN. If you specify only the function name, it is limited to 64 character in length.

Length constraints: Minimum length of 1. Maximum length of 111.

Pattern:
`(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)`

Request Body

The request requires the following data in JSON format.

S3Bucket
Amazon S3 bucket name where the .zip file containing your deployment package is stored. This bucket must reside in the same AWS region where you are creating the Lambda function.

Type: String

Length constraints: Minimum length of 3. Maximum length of 63.

Pattern: `^[0-9A-Za-z\.\-_]*(?<!\.)$`

Required: No

S3Key
The Amazon S3 object (the deployment package) key name you want to upload.

Type: String

Length constraints: Minimum length of 1. Maximum length of 1024.

Required: No

S3ObjectVersion
The Amazon S3 object (the deployment package) version you want to upload.

Type: String

Length constraints: Minimum length of 1. Maximum length of 1024.

Required: No

ZipFile
Based64-encoded .zip file containing your packaged source code.

Type: Blob

Required: No

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "CodeSize": number,
    "Description": "string",
    "FunctionArn": "string",
    "FunctionName": "string",
    "Handler": "string",
    "LastModified": "string",
    "MemorySize": number,
    "Role": "string",
    "Runtime": "string",
    "Timeout": number
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

CodeSize
The size, in bytes, of the function .zip file you uploaded.

Type: Long

Description
The user-provided description.

Type: String

Length constraints: Minimum length of 0. Maximum length of 256.

FunctionArn
The Amazon Resource Name (ARN) assigned to the function.

Type: String

Pattern: arn:aws:lambda:[a-z]{2}-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(\/[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12})?

FunctionName
The name of the function.

Type: String

Length constraints: Minimum length of 1. Maximum length of 111.

Pattern: `(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)`

Handler
The function Lambda calls to begin executing your function.

Type: String

Length constraints: Minimum length of 0. Maximum length of 128.

Pattern: `[^\s]+`

LastModified
The timestamp of the last time you updated the function.

Type: String

MemorySize
The memory size, in MB, you configured for the function. Must be a multiple of 64 MB.

Type: Number

Valid range: Minimum value of 128. Maximum value of 1536.

Role
The Amazon Resource Name (ARN) of the IAM role that Lambda assumes when it executes your function to access any other Amazon Web Services (AWS) resources.

Type: String

Pattern: `arn:aws:iam::\d{12}:role/?[a-zA-Z_0-9+=,.@\-_/]+`

Runtime
The runtime environment for the Lambda function.

Type: String

Valid Values: `nodejs | java8`

Timeout
The function execution time at which Lambda should terminate the function. Because the execution time has cost implications, we recommend you set this value based on your expected execution time. The default is 3 seconds.

Type: Number

Valid range: Minimum value of 1. Maximum value of 60.

Errors

CodeStorageExceededException
HTTP Status Code: 400

InvalidParameterValueException
One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

ResourceNotFoundException
The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 400

ServiceException
The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException
HTTP Status Code: 400

UpdateFunctionConfiguration

Updates the configuration parameters for the specified Lambda function by using the values provided in the request. You provide only the parameters you want to change. This operation must only be used on an existing Lambda function and cannot be used to update the function's code.

This operation requires permission for the `lambda:UpdateFunctionConfiguration` action.

Request Syntax

```
PUT /2015-03-31/functions/FunctionName/versions/HEAD/configuration HTTP/1.1
Content-type: application/json

{
    "Description": "string",
    "Handler": "string",
    "MemorySize": number,
    "Role": "string",
    "Timeout": number
}
```

URI Request Parameters

The request requires the following URI parameters.

FunctionName
The name of the Lambda function.

You can specify an unqualified function name (for example, "Thumbnail") or you can specify Amazon Resource Name (ARN) of the function (for example, "arn:aws:lambda:us-west-2:account-id:function:ThumbNail"). AWS Lambda also allows you to specify only the account ID qualifier (for example, "account-id:Thumbnail"). Note that the length constraint applies only to the ARN. If you specify only the function name, it is limited to 64 character in length.

Length constraints: Minimum length of 1. Maximum length of 111.

Pattern:
`(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)`

Request Body

The request requires the following data in JSON format.

Description
A short user-defined function description. AWS Lambda does not use this value. Assign a meaningful description as you see fit.

Type: String

Length constraints: Minimum length of 0. Maximum length of 256.

Required: No

Handler
The function that Lambda calls to begin executing your function. For Node.js, it is the *module-name.export* value in your function.

Type: String

Length constraints: Minimum length of 0. Maximum length of 128.

Pattern: `[^\s]+`

Required: No

MemorySize

The amount of memory, in MB, your Lambda function is given. AWS Lambda uses this memory size to infer the amount of CPU allocated to your function. Your function use-case determines your CPU and memory requirements. For example, a database operation might need less memory compared to an image processing function. The default value is 128 MB. The value must be a multiple of 64 MB.

Type: Number

Valid range: Minimum value of 128. Maximum value of 1536.

Required: No

Role

The Amazon Resource Name (ARN) of the IAM role that Lambda will assume when it executes your function.

Type: String

Pattern: `arn:aws:iam::\d{12}:role/?[a-zA-Z_0-9+=,.@\-_/]+`

Required: No

Timeout

The function execution time at which AWS Lambda should terminate the function. Because the execution time has cost implications, we recommend you set this value based on your expected execution time. The default is 3 seconds.

Type: Number

Valid range: Minimum value of 1. Maximum value of 60.

Required: No

Response Syntax

```
HTTP/1.1 200
Content-type: application/json

{
    "CodeSize": number,
    "Description": "string",
    "FunctionArn": "string",
    "FunctionName": "string",
    "Handler": "string",
    "LastModified": "string",
    "MemorySize": number,
    "Role": "string",
    "Runtime": "string",
    "Timeout": number
}
```

Response Elements

If the action is successful, the service sends back an HTTP 200 response.

The following data is returned in JSON format by the service.

CodeSize
The size, in bytes, of the function .zip file you uploaded.

Type: Long

Description
The user-provided description.

Type: String

Length constraints: Minimum length of 0. Maximum length of 256.

FunctionArn
The Amazon Resource Name (ARN) assigned to the function.

Type: String

Pattern: arn:aws:lambda:[a-z]{2}-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(\V[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12})?

FunctionName
The name of the function.

Type: String

Length constraints: Minimum length of 1. Maximum length of 111.

Pattern: `(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)`

Handler
The function Lambda calls to begin executing your function.

Type: String

Length constraints: Minimum length of 0. Maximum length of 128.

Pattern: `[^\s]+`

LastModified
The timestamp of the last time you updated the function.

Type: String

MemorySize
The memory size, in MB, you configured for the function. Must be a multiple of 64 MB.

Type: Number

Valid range: Minimum value of 128. Maximum value of 1536.

Role
The Amazon Resource Name (ARN) of the IAM role that Lambda assumes when it executes your function to access any other Amazon Web Services (AWS) resources.

Type: String

Pattern: `arn:aws:iam::\d{12}:role/?[a-zA-Z_0-9+=,.@\-_/]+`

Runtime
The runtime environment for the Lambda function.

Type: String

Valid Values: `nodejs | java8`

Timeout
The function execution time at which Lambda should terminate the function. Because the execution time has cost implications, we recommend you set this value based on your expected execution time. The default is 3 seconds.

Type: Number

Valid range: Minimum value of 1. Maximum value of 60.

Errors

InvalidParameterValueException
One of the parameters in the request is invalid. For example, if you provided an IAM role for AWS Lambda to assume in the `CreateFunction` or the `UpdateFunctionConfiguration` API, that AWS Lambda is unable to assume you will get this exception.

HTTP Status Code: 400

ResourceNotFoundException
The resource (for example, a Lambda function or access policy statement) specified in the request does not exist.

HTTP Status Code: 400

ServiceException
The AWS Lambda service encountered an internal error.

HTTP Status Code: 500

TooManyRequestsException
HTTP Status Code: 400

Data Types

The AWS Lambda API contains several data types that various actions use. This section describes each data type in detail.

Note
The order of each element in the response is not guaranteed. Applications should not assume a particular order.

The following data types are supported:

- EventSourceMappingConfiguration (p. 200)
- FunctionCode (p. 201)
- FunctionCodeLocation (p. 202)
- FunctionConfiguration (p. 202)

EventSourceMappingConfiguration

Description

Describes mapping between an Amazon Kinesis stream and a Lambda function.

Contents

BatchSize
The largest number of records that AWS Lambda will retrieve from your event source at the time of invoking your function. Your function receives an event with all the retrieved records.

Type: Number

Valid range: Minimum value of 1. Maximum value of 10000.

Required: No

EventSourceArn
The Amazon Resource Name (ARN) of the Amazon Kinesis stream that is the source of events.

Type: String

Pattern: `arn:aws:([a-zA-Z0-9\-])+:([a-z]{2}-[a-z]+-\d{1})?:(\d{12})?:(.*)`

Required: No

FunctionArn
The Lambda function to invoke when AWS Lambda detects an event on the stream.

Type: String

Pattern:
arn:aws:lambda:[a-z]{2}-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(\/[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12})?

Required: No

LastModified
The UTC time string indicating the last time the event mapping was updated.

Type: DateTime

Required: No

LastProcessingResult
The result of the last AWS Lambda invocation of your Lambda function.

Type: String

Required: No

State
The state of the event source mapping. It can be "Creating", "Enabled", "Disabled", "Enabling", "Disabling", "Updating", or "Deleting".

Type: String

Required: No

StateTransitionReason
The reason the event source mapping is in its current state. It is either user-requested or an AWS Lambda-initiated state transition.

Type: String

Required: No

UUID
The AWS Lambda assigned opaque identifier for the mapping.

Type: String

Required: No

FunctionCode

Description

The code for the Lambda function.

Contents

S3Bucket
Amazon S3 bucket name where the .zip file containing your deployment package is stored. This bucket must reside in the same AWS region where you are creating the Lambda function.

Type: String

Length constraints: Minimum length of 3. Maximum length of 63.

Pattern: `^[0-9A-Za-z\.\-_]*(?<!\.)$`

Required: No

S3Key
The Amazon S3 object (the deployment package) key name you want to upload.

Type: String

Length constraints: Minimum length of 1. Maximum length of 1024.

Required: No

S3ObjectVersion
The Amazon S3 object (the deployment package) version you want to upload.

Type: String

Length constraints: Minimum length of 1. Maximum length of 1024.

Required: No

ZipFile
A base64-encoded .zip file containing your deployment package. For more information about creating a .zip file, go to Execution Permissions in the *AWS Lambda Developer Guide*.

Type: Blob

Required: No

FunctionCodeLocation

Description

The object for the Lambda function location.

Contents

Location
The presigned URL you can use to download the function's .zip file that you previously uploaded. The URL is valid for up to 10 minutes.

Type: String

Required: No

RepositoryType
The repository from which you can download the function.

Type: String

Required: No

FunctionConfiguration

Description

A complex type that describes function metadata.

Contents

CodeSize
The size, in bytes, of the function .zip file you uploaded.

Type: Long

Required: No

Description
The user-provided description.

Type: String

Length constraints: Minimum length of 0. Maximum length of 256.

Required: No

FunctionArn
The Amazon Resource Name (ARN) assigned to the function.

Type: String

Pattern: arn:aws:lambda:[a-z]{2}-[a-z]+-\d{1}:\d{12}:function:[a-zA-Z0-9-_]+(\:[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12})?

Required: No

FunctionName
The name of the function.

Type: String

Length constraints: Minimum length of 1. Maximum length of 111.

Pattern:
`(arn:aws:lambda:)?([a-z]{2}-[a-z]+-\d{1}:)?(\d{12}:)?(function:)?([a-zA-Z0-9-_]+)`

Required: No

Handler
The function Lambda calls to begin executing your function.

Type: String

Length constraints: Minimum length of 0. Maximum length of 128.

Pattern: `[^\s]+`

Required: No

LastModified
The timestamp of the last time you updated the function.

Type: String

Required: No

MemorySize
The memory size, in MB, you configured for the function. Must be a multiple of 64 MB.

Type: Number

Valid range: Minimum value of 128. Maximum value of 1536.

Required: No

Role
The Amazon Resource Name (ARN) of the IAM role that Lambda assumes when it executes your function to access any other Amazon Web Services (AWS) resources.

Type: String

Pattern: `arn:aws:iam::\d{12}:role/?[a-zA-Z_0-9+=,.@\-_/]+`

Required: No

Runtime
The runtime environment for the Lambda function.

Type: String

Valid Values: `nodejs | java8`

Required: No

Timeout
The function execution time at which Lambda should terminate the function. Because the execution time has cost implications, we recommend you set this value based on your expected execution time. The default is 3 seconds.

Type: Number

Valid range: Minimum value of 1. Maximum value of 60.

Required: No

Document History

The following table describes the important changes to the *AWS Lambda Developer Guide*.

Relevant Dates to this History:

- **Current product version**: 2015-03-31
- **Last documentation update**: August 27, 2015

Change	Description	Date
Two new walkthroughs	The following new walkthroughs are added. They both use Java Lambda function. AWS Lambda Walkthrough 3: Process Amazon DynamoDB Events (Java) (p. 122) AWS Lambda Walkthrough 4: Handling Mobile User Application Events for Android (Java) (p. 124)	In this release
Support for DynamoDB Streams	DynamoDB Streams is now generally available and you can use it in all the regions where DynamoDB is available. You can enable DynamoDB Streams for your table and use a Lambda function as a trigger for the table. Triggers are custom actions you take in response to updates made to the DynamoDB table. For an example walkthrough, see AWS Lambda Walkthrough 3: Processing Events from Amazon DynamoDB Streams Using the AWS CLI (Node.js) (p. 53).	July 14, 2015

Change	Description	Date
AWS Lambda now supports invoking Lambda functions with REST-compatible clients.	Until now, to invoke your Lambda function from your web, mobile, or IoT application you needed the AWS SDKs (for example, AWS SDK for Java, AWS SDK for Android, or AWS SDK for iOS). Now, AWS Lambda supports invoking a Lambda function with REST-compatible clients through a customized API that you can create using Amazon API Gateway. You can send requests to your Lambda function endpoint URL. You can configure security on the endpoint to allow open access, leverage AWS Identity and Access Management (IAM) to authorize access, or use API keys to meter access to your Lambda functions by others. For an example Getting Started exercise, see Getting Started 4: Creating an HTTP Endpoint-Enabled Lambda Function (AWS Lambda Integration with Amazon API Gateway) (p. 27). For more information about the Amazon API Gateway, go to http://aws.amazon.com/api-gateway/.	July 09, 2015
The AWS Lambda console now provides blueprints to easily create Lambda functions and test them.	AWS Lambda console provides a set of *blueprints*. Each blueprint provides a sample event source configuration and sample code for your Lambda function that you can use to easily create Lambda-based applications. All of the AWS Lambda Getting Started exercises now use the blueprints. For more information, see Getting Started: Authoring AWS Lambda Code in Node.js (p. 15).	In this release
AWS Lambda now supports Java to author your Lambda functions.	You can now author Lambda code in Java. For more information, see Authoring Lambda Functions in Java (p. 88).	June 15, 2015
AWS Lambda now supports specifying an Amazon S3 object as the function .zip when creating or updating a Lambda function.	You can upload a Lambda function deployment package (.zip file) to an Amazon S3 bucket in the same region where you want to create a Lambda function. Then, you can specify the bucket name and object key name when you create or update a Lambda function.	May 28, 2015

Change	Description	Date
AWS Lambda now generally available with added support for mobile backends	AWS Lambda is now generally available for production use. The release also introduces new features that make it easier to build mobile, tablet, and Internet of Things (IoT) backends using AWS Lambda that scale automatically without provisioning or managing infrastructure. AWS Lambda now supports both real-time (synchronous) and asynchronous events. Additional features include easier event source configuration and management. The permission model and the programming model have been simplified by the introduction of resource policies for your Lambda functions. The documentation has been updated accordingly. For information, see the following topics: AWS Lambda: How it Works (p. 3) Getting Started: Authoring AWS Lambda Code in Node.js (p. 15) http://aws.amazon.com/lambda/whatsnew	April 9, 2015
Preview release	Preview release of the *AWS Lambda Developer Guide*.	November 13, 2014

AWS Glossary

For the latest AWS terminology, see the AWS Glossary in the *AWS General Reference*.